Message Passing with Communication Structures

Gagarine Yaikhom

Doctor of Philosophy
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2006

Abstract

Abstraction concepts based on process groups have largely dominated the design and implementation of communication patterns in message passing systems. Although such an approach seems pragmatic—given that participating processes form a 'group'—in this dissertation, we discuss subtle issues that affect the qualitative and quantitative aspects of this approach. To address these issues, we introduce the concept of a 'communication structure,' which defines a communication pattern as an implicit runtime composition of localised patterns, known as 'roles.' During application development, communication structures are derived from the algorithm being implemented. These are then translated to an executable form by defining process specific data structures, known as 'branching channels.'

The qualitative advantages of the communication structure approach are that the resulting programming model is non-ambiguous, uniform, expressive, and extensible. To use a pattern is to access the corresponding branching channels; to define a new pattern is simply to combine appropriate roles. The communication structure approach therefore allows immediate implementation of ad hoc patterns. Furthermore, it is guaranteed that every newly added role interfaces correctly with all of the existing roles, therefore scaling the benefit of every new addition.

Quantitatively, branching channels improve performance by automatically overlapping computations and communications. The runtime system uses a receiver initiated communication protocol that allows senders to continue immediately without waiting for the receivers to respond. The advantage is that, unlike split-phase asynchronous communications, senders need not check whether the send operations were successful. Another property of branching channels is that they allow communications to be grouped, identified, and referenced. Communication structure specific parameters, such as message buffering, can therefore be specified immediately. Furthermore, a 'commit' based interface optimisation for send-and-forget type communications—where senders do not reuse sent data—is presented. This uses the referencing property of branching channels, allowing message buffering without incurring performance degradation due to intermediate memory copy.

Acknowledgements

I would like to thank my supervisor Dr. Murray Cole for being very supportive and enthusiastic about the work presented in this dissertation. He not only gave me the freedom to explore potential avenues for research, but also kept me on track by analysing my arguments and approach critically. I am grateful to him for all the things that I have learned under his supervision.

Thanks are also due to Dr. Michael O'Boyle (second supervisor) and Dr. Marcelo Cintra for their interesting comments during the annual progress meetings.

I would like to thank my family for supporting me throughout my life in every possible way. Special thanks go to my parents for providing me with every academic resource that I needed, even during financial hardships. I have reached this far mainly because of their support and enthusiasm for education. I hope that my work is worth some of the sacrifices which they have made for my siblings and I.

I would also like to thank Horacio González-Vélez, John Hawkins, Robert Hutchison, and Samantha Lyle for being great friends. They made sure that I did not neglect my social life entirely. In particular, I would like to thank Horacio for the interesting discussions we had about research, computer science, and life in general.

Last, but not least, I would like to thank Marie Melvin for providing me with emotional support throughout the period of my research. She also helped me during the writing process by checking the drafts for typographical and grammatical errors. Without her support, it would have been a lonely experience.

This research was funded by the Overseas Research Studentship (ORS) award, and the University of Edinburgh scholarships.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. The following articles were published during the course of this research,

- Gagarine Yaikhom, Shared Message Buffering without Intermediate Memory Copy, *In Proc. of the* 3rd *Intl. Workshop on Higher-level Parallel Programming and Applications (HLPP)*, July 2005, Warwick, United Kingdom. (Also to appear in *Parallel Processing Letters*)
- Gagarine Yaikhom, Buffered Branching Channels with *Rendezvous* Message Passing, *In Proc. of the* 23rd *IASTED Intl. Conf. on Parallel and Distributed Computing and Networks (PDCN)*, pages 184–192, ACTA Press, February 2005, Innsbruck, Austria.

(*Gagarine Yaikhom*)

Dedicated to my beloved parents

Late. L. Bhanumati Devi

and

Yaikhom Tomcha Singh.

Contents

CHAPTER 1

Introduction

MESSAGE PASSING PARALLEL PROGRAMS have become a practical reality with the advent of loosely-coupled parallel or distributed systems such as the network of workstations (e.g. the Beowulf cluster). Loosely-coupled systems provide an array of processors which can be used simultaneously for a computational task. None of these processors, however, share a physical memory space; the data dependencies between processors are therefore satisfied by explicitly passing messages over a communications network. Due to the proven scalability and cost effectiveness of loosely-coupled systems, interest in such systems has grown significantly over the years [96, 58].

When a new computing system is introduced, various abstraction models are also introduced. These models define abstraction concepts that can be related directly to the underlying system components. The interesting feature of abstraction based programming is that, through a suitable programming model, the low-level implementation details of the abstract concepts can be concealed. This allows programmers to easily harness the facilities provided by a system for useful computations without actually understanding every low-level implementation detail. The design of an abstraction model, however, is usually based on different conceptual interpretations of a given system. Consequently, the level of abstraction provided by different abstraction models differs widely. Most models, therefore, aim to assist the programmer by providing a proper balance between efficiency and programmability.

In this chapter, we discuss the motivations behind this dissertation; followed by a statement of the thesis, and the approach undertaken. We then discuss the contributions made by this dissertation; followed by an outline of the contents of the remaining chapters. Finally, this chapter concludes with a description of the mathematical notations and pseudocode conventions followed throughout.

1.1 Motivation

In message passing systems, abstraction concepts based on *process groups* have largely dominated the design and implementation of communication patterns. Although such an approach seems pragmatic—given that participating processes form a 'group'—in this chapter, we discuss subtle issues that affect the qualitative and quantitative aspects of programming with process group based abstraction models.

When a message passing program is developed, the complexity of programming is mainly concentrated in the code segments that represent data communications between the processing elements (hereafter referred to as *processes*). An increase in the complexity of a message passing algorithm is usually accompanied by an increase in the complexity of these inter-process communications. In order to reduce the programming effort, a message passing abstraction model therefore provides the programmer with concepts that can be related directly to these inter-process communications.

In general, inter-process communications can be viewed as a producer-consumer relationship: producer processes send data that have been produced or transformed; these are then received, and consumed, by consumer processes during further computations. Because such producer-consumer relationships are inherent in message passing algorithms, they often form the basis for most of the popular abstraction models. Consequently, representing a *point-to-point* communication in the corresponding programming model is pretty straightforward: it is often expressed with some form of a *send-receive* pair [6]. When communications involving more than two processes—usually referred to as *collective communications*—are considered, however, abstraction models based on 'process groups' raise subtle programming issues.

A process group defines a logical grouping of processes which is used to derive a collective communication domain. In popular programming models, such as the MPI [93], abstraction for collective communications is based on this concept of a *process group* [14]. From a given process group several collective communication domains can be derived, which are often associated with an opaque data structure known as the *communicator*. When a collective communication is performed over a communicator, every process in the corresponding process group participates in that communication. This approach raises some programming issues that we shall introduce in light of the following example:

Example 1.1.1
Assume a collective communication involving five processes: A (accountant), R (re-

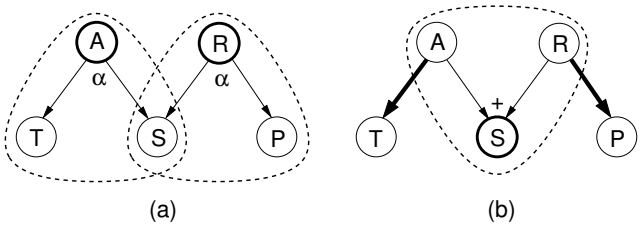


Figure 1.1: Decomposition of an overlapping communication domain. (a) Decomposition based on two *scatters* (α) on groups $\{A, T, S\}$ and $\{R, S, P\}$, where A and R are the roots of the scatter. (b) Decomposition based on one *reduce* ($+$) on group $\{A, S, R\}$, where S is the destination of the reduced value.

search council), T (teacher), S (student), and P (research project). The aim is to simulate the financial transactions for a period of 12 months, where at the end of each month A sends funds to T and S, and R sends funds to S and P. S receives funds from A and R, T receives funds from A, and P receives funds from R. Each of these monthly communications is a collective communication because all of the five processes—A, R, T, S and P—should participate during that communication. ▌

Two collective communication domains are said to be *overlapping* if there is at least one process which is associated with both domains, so that a single collective communication cannot be invoked on the common process. In Example 1.1.1, if we consider the scattering[1] of data from A and R, and sum reduction[2] of data on S, we can see that the single sum reduction communication on S actually represents part of both scatters from A and R. This means that the scatter groups in A and R are overlapping because of the common receiver S. In the same scenario, if we ignore the sum reduction and decompose that sum reduction into two receive communications, where each communication is separately associated with the scattering on A and R, then the two collective communication domains defined by A and R do not overlap as each of the two communications on S are separately performed for each communication domain.

A necessary condition for using collective communications is that they should not be performed on overlapping communication domains. In such cases, the communication domain should be decomposed into multiple non-overlapping communication domains. Such decompositions, however, raise the following issues:

[1] *Scatter* is a communication pattern which involves sending data from the root process (the sender) to the receiver processes in a process group so that every receiver receives a unique data.

[2] *Sum reduction* is a communication pattern where unique data from multiple sender processes, in a process group, is reduced on the root process (the receiver) so that the receiver receives the sum of all the values sent by the senders. Such patterns integrate trivial computations with the communications.

1. What is the approach for carrying out the decomposition? For Example 1.1.1, a brief inspection reveals that the overlapping domains can be decomposed in two ways, as shown in Figure 1.1. In the first decomposition (Figure 1.1.a), the overlapping domain is decomposed by choosing two collective communications over the groups $\{A, T, S\}$ and $\{R, S, P\}$. These communications emphasise the scattering (α) of funds from A and R respectively. In the second decomposition (Figure 1.1.b), the overlapping domain is decomposed through a single collective communication over the group $\{A, S, R\}$, emphasising the sum reduction ($+$) at S: which also introduces two point-to-point communications from A to T, and R to P (shown with thick arrows). Both decompositions yield valid collective communications; however, because of the resulting ambiguity due to the possible choices, performing an optimal decomposition that should also provide a reasonably efficient performance may aggravate the already complex programming exercise. We shall refer to this issue as *the ambiguity problem*.
2. If we assume that the overlapping communication domains have been decomposed somehow, how does a programmer choose a particular decomposition? Should the decisions be made based on the qualitative aspects, such as programming simplicity, or the quantitative aspects, such as better performance? If qualitative aspects are to be considered most crucial, how does a programmer compare the decompositions qualitatively? On the other hand, if quantitative aspects are to be considered crucial, how does a programmer decide which decomposition will yield the best performance, without understanding the underlying implementation of the message passing interfaces? Furthermore, if a programmer is allowed to understand the underlying implementation details of the message passing interfaces, the message passing model has then failed to provide a proper abstraction—abstraction models should conceal such details. We shall refer to this issue as *the choice dilemma*.
3. Another drawback of choosing decompositions based on the implementation details of the underlying run-time system is that the performance of a program can no longer be considered portable (usually referred to as *performance portability* [55]). This is because, in practice, there exist many implementations for a given abstraction model, which are often based on widely varying design decisions. A given decomposition may therefore perform better than the other decompositions on a particular implementation of the abstraction model; however, it may also perform poorly on other implementations of the abstraction model. We shall refer to this issue as *the performance portability problem*.
4. A qualitative issue related to structured parallel programming is the sacrifice of

'structural' information due to the decomposition. When an overlapping communication domain is decomposed, information that enhances the comprehensibility of the algorithm is also sacrificed. For example, if the first decomposition of Example 1.1.1 is chosen, the information about the sum reduction on S is lost. In the program, therefore, instead of invoking a single sum reduction communication interface, S should invoke two interfaces for the scatters from A and R, following which summation of the received funds is performed separately. Alternatively, if the second decomposition is chosen, the information about the scatters from A and R is lost. The processes A and R, therefore, must invoke sum reduction and point-to-point communication interfaces instead of invoking a single scatter interface.

Such sacrifice of information seriously affects the programming, and maintenance, of parallel message passing programs because once the program is developed, there is no way of describing the previous algorithm 'completely' without making certain assumptions. For example, the separate summation of funds required by the first decomposition now constitutes a code segment of its own, and should therefore be treated separately from the communications. Even though this implementation also expresses a semantically equivalent representation, it is not clear that S could have simplified the implementation by using a sum reduction communication interface. We shall refer to this issue as *the loss of structural information problem*.

5. Another qualitative issue related to collective communications based on process groups is that processes are required to acknowledge other processes with which they do not directly share a producer-consumer relationship. For example, in the first decomposition of Example 1.1.1, T and S are required to mutually acknowledge each other during the scatter (because of the process group $\{A, T, S\}$), when they do not actually communicate data in any sense. This means that, in order to receive the monthly salary, the teacher is required to know about all of the other entities who are also receiving funds from the accountant (in this case the student); the same goes for the student. Similar arguments can be applied to the process group $\{R, S, P\}$. In daily practice, such systems would be considered impractical because they demand details that are irrelevant to the execution of a given process. We shall refer to this issue as *the redundant acknowledgement problem*.

This dissertation develops an abstraction model that resolves the above issues. In addition to resolving these issues, the abstraction model allows implementation

of a programming model that is non-ambiguous, uniform, expressive, extensible, and yet efficient. By redefining the meaning of a collective communication, and collective communication patterns, the new programming model allows communication specific optimisations, such as specialised message buffering and message buffering without intermediate memory copy, ad hoc implementation of communication patterns, and single phase asynchronous communications.

1.2 Thesis and approach

The thesis of this dissertation is that message passing abstraction models should emphasise structured programming of inter-process communication patterns, just like data structures are used in structured sequential programming. The approach for performing such structured programming of communication patterns, which we suggest in this dissertation, is based on our argument that holistic communication patterns are best abstracted as an implied runtime composition of process specific localised communication patterns.

Abstraction models should not dictate how a message passing algorithm is implemented, and therefore should not suffer from the issues discussed previously. They should, instead, provide the necessary mechanisms to effortlessly realise any given communication pattern manifested by the algorithm. The term 'effortless' refers to the qualitative properties—non-ambiguity, expressiveness, uniformity, and extensibility—defined (see also Table 6.1) as follows:

1. **Non-ambiguity.** An abstraction model, and therefore the corresponding programming model, is said to be non-ambiguous if it does not allow implementation of a given communication pattern into different application programs with different characteristics (e.g. performance, programming complexity etc.) based on the different interpretations of the communication pattern allowed by the abstraction model.
2. **Expressiveness.** A programming model is said to be expressive if a communication pattern manifested by an algorithm can be implemented with the application programming interfaces without adapting the communication pattern to fit the programming model.
3. **Uniformity.** A programming model is said to be uniform if the application programming interfaces that it provides are uniform in terms of the interface function prototype (e.g. number of parameters, data type of parameters etc.), and the manner in which the communication pattern is expressed through these interfaces within an application program.

4. **Extensibility.** A programming model is said to be extensible if it allows extension of the existing programming model with additional facilities (e.g. new communication patterns etc.) without disrupting the usage and the functioning of the existing application programming interfaces.

In addition to having the qualitative properties discussed above, the model should be practically implementable, with performance characteristics reasonably close to, or better than, the existing message passing systems.

In the suggested abstraction model, collective communications—and therefore inter-process communication patterns—are not abstracted based on the concept of a process group. They are instead abstracted implicitly as run-time compositions of process specific communication components, which are defined based on the process specific understanding of the communication pattern manifested by the algorithm which is being implemented. Furthermore, to generalise the concepts to all forms of communication, point-to-point communications are considered to be a special case: a collective communication with only two processes.

1.3 Contribution and outline

The contributions of this dissertation can be summarised as follows:

The first contribution of this dissertation constitutes development of the abstraction model which resolves the issues discussed in Section 1.1. We do this by defining new concepts which are free from the notion of a process group. In essence, we begin with the control flow graph of a sequential program, and build up the theory from these foundations into an abstract representation of a message passing parallel program.

The second contribution is the introduction of the β-channel programming model based on the abstraction model. We show that the programming model facilitates 'effortless' message passing programming with respect to the following four qualities: non-ambiguity, expressiveness, uniformity, and extensibility.

The third contribution concerns implementation of higher-level programming constructs, where we use β-channels for practical skeletal programming. By implementing Cole's *algorithmic skeletons* [32, 33], we demonstrate how implementation and deployment of skeletal programming can be simplified with the new programming model.

The fourth, and final, contribution is related to the performance aspect of the programming model. The programming model introduces a method for specialised message buffering, which allows selective buffering for specific communications.

This results in the flexibility to implement a single phase asynchronous *rendezvous* communication protocol, which can be extended further as an optimisation allowing message buffering without incurring performance degradation due to intermediate memory copy. We show empirically that the new approach improves the overall performance of an application by automatically increasing the overlapping of computations and communications within the application.

The rest of the dissertation is organised as follows:

In Chapter 2, we survey existing approaches that are currently used for message passing programming. This survey focuses on the practical implementation of inter-process communications, and communication patterns. We also discuss the psychological aspects of programming and system design, which provide supporting arguments for the design decisions that are made in Chapter 3.

In Chapter 3, we develop the abstraction model. The arguments discussed in Chapter 2 form the guiding principles for these conceptual developments. We define abstract concepts based on our thesis which represents a communication pattern as the implied runtime composition of process specific communication components. Throughout this chapter, we discuss how the new model resolves the issues discussed in Section 1.1. Additionally, we discuss the properties of the new model which offer several practical advantages, such as avoiding intermediate memory copy during buffering, communication specific specialised message buffering, automatic overlapping of computations and communications etc..

In Chapter 4, we describe the programming model. A two-phase application development process is suggested, followed by a description of the application programming interfaces. As a demonstration, these programming interfaces are then used to implement several message passing programs with widely varying communication patterns. The final part of this chapter explores the relationship between the β-channel programming model and skeletal parallel programming.

In Chapter 5, we discuss the low-level implementation details of the programming model. We describe the multi-threaded runtime system, the manner in which application programs are executed, and the communication protocol that is used to transfer messages. This is followed by a discussion on the integration of specialised message buffering, and the optimisation for avoiding intermediate memory copy.

In Chapter 6, we evaluate the qualitative and quantitative characteristics of the new approach. And finally, in Chapter 7, we conclude the dissertation and suggest areas for further research.

1.4 Notations and conventions

This section describes the mathematical notations and the pseudocode conventions that are followed throughout this dissertation.

The mathematical notations are,

$\mathcal{P}_i$	Process with rank i.
$\lceil x \rceil$	Smallest integer not less than x.
$\lfloor x \rfloor$	Largest integer not greater than x.
$\forall x$	For all x.
$\exists x$	There exists x.
$\lvert x \rvert$	Absolute value of real number x.
$\emptyset$	Empty set.
$\lvert S \rvert$	Cardinality of set S.
$x \in S$	x is an element of set S.
$x \notin S$	x is not an element of set S.
$X \cap Y$	Set intersection of X and Y.
$X \cup Y$	Set union of X and Y.
$X \setminus Y$	Set difference of X and Y.
$\vee$	Logical conjunction.
$\wedge$	Logical disjunction.
$\neg$	Logical negation.

We follow a pseudocode convention similar to Fraser's [42]. Pseudocodes are expressed with the C [68] programming language: except for the following operator substitutions that are made for clarity,

	C language	Pseudocode
Assignment	=	:=
Bitwise	<<, >>, ^, ~	$\ll$, $\gg$, $\otimes$, ~
Equality	==, !=	$=$, $\neq$
Logical	&&, ||, !	$\wedge$, $\vee$, $\neg$
Member pointer	−>	$\rightarrow$
Relational	<=, >=	$\leqslant$, $\geqslant$

CHAPTER 2

Background

In this chapter we place our work in the context of existing and ongoing work. In Section 2.1 we discuss what this dissertation is about. We provide a general introduction to our subject, and present an overview of the areas which we plan to address in this dissertation. In Section 2.2, we explore existing and ongoing work in the field. We discuss the different approaches that have been suggested so far, while strongly emphasising the models that have proved successful in recent years. In Section 2.3, we discuss the objectives of this dissertation. Finally, in Section 2.4 we discuss the approach with which we plan to attain our objectives.

2.1 What is this dissertation about?

Since their advent, digital computing systems have become an integral part of our civilisation. They are now being used intensively in academic institutions, commercial institutions, and institutions related to art and entertainment. The ever increasing demand for high performance computing, however, comes from scientific research where computing systems are being used to model our physical world. Some of these advanced applications are related to simulation of physical models, such as weather forecasting, or the extraction of information from a large data set, such as DNA sequencing, or the analysis of astronomical data. With such applications, the demand for high performance computing has reached the stage where sequential computing systems can no longer provide the necessary computing power. As a result, interest in the field of parallel computing systems has grown significantly over the years.

So far, many parallel computing systems have been introduced [96]. Although most of these systems are based on different design decisions, they can be broadly classified into two categories: (1) shared memory systems, and (2) distributed memory systems. In shared memory systems (also referred to as *tightly-coupled* systems) all the processes in the system share a common memory. On the other hand, in distributed memory systems (also referred to as *loosely-coupled* systems) all the processes do not share a common memory; instead, every process is associated with its own memory subsystem. Transfer of data from one process to another is therefore performed through data communications over a backbone network.

Recently, two types of distributed computing systems have become prominent: (1) Cluster computing systems [26], and (2) the Grid computing environments [50]. The *Cluster* is a network of workstations, which is cost effective, and highly scalable. The *Grid* is similar to a cluster but entails a more dynamic environment where computing resources, in geographically diverse locations, can enter or leave the environment. Since Beowulf clusters are more widely available than specialised parallel computers, our aim is to explore approaches that can be used to program these systems easily and efficiently.

This dissertation is about designing a message passing abstraction model which can be expressed with a parallel programming model that is non-ambiguous, uniform, expressive, and extensible. Not only should the programming model be simple to use, but it should also provide a runtime system which performs better than, or is at least comparable to, the existing ones.

In the next section, we explore existing approaches to parallel programming.

2.2 What are the existing approaches?

In this section, we explore existing approaches for parallel programming, and emphasise message passing programming models in particular. For a more exhaustive survey of the field, see Andrews and Schneider [7], Skillicorn and Talia [92], Foster [40], Quinn [86], Leopold [71], and Grama *et al*. [49].

Leopold defines parallel and distributed computing as follows:

> "Parallel computing splits an application up into tasks that are executed *at the same time*, whereas distributed computing splits an application up into tasks that are executed *at different locations*, using *different resources*." [71, page 3]

From the above definition, one can infer that parallel computing is about dividing a computational task into sub-tasks that can be computed simultaneously using different computing resources.

Based on the manner in which a parallel program is developed, the development process is usually classified into two styles: (1) *single program multiple data* (SPMD), and (2) *multiple program multiple data* (MPMD) [71, page 25–26]. In the SPMD style, a single program is developed, which is executed by all the processes. The execution instances of the program on any two processes are, however, not necessarily the same. In the MPMD style, different programs are developed for different processes, so that differences in the execution instance depend on the program which a process executes. The SPMD style is normally used to implement algorithms with *data parallelism* [71, Chapter 3]; MPMD style, on the other hand, is more appropriate for algorithms with *task parallelism* [71, page 53].[1] In distributed systems, such as the network of workstations, the SPMD style is considered to be more appropriate [76, page 71], one popular system being the MPI [93].

Some of the popular programming models for distributed systems are:

Message passing models In a *message passing* model, all the processes in the computing system are considered to be peers. Unless otherwise specified within the application program, all the processes are considered to be the same with regard to their capabilities to execute a given task. During parallel execution, these processes are coordinated in a manner that will allow them to participate in a single computation simultaneously. When data from one process is required in another, the data is communicated directly from the sender to the receiver by performing message passing operations on both processes.

[1] *Task parallelism* and *data parallelism* are sometimes defined in different ways. In this dissertation, by task parallelism we mean different tasks that can be executed simultaneously; by data parallelism we mean different data sets that can be processed simultaneously.

Client-server models In a *client-server* model [71, Chapter 6], a client process requests service from a server. The server can be either an active node, which performs computations for the client, or a passive node, which manages computation on various clients (for example, a data server). In practice, client-server models tend to be more passive (for example, a web server). The interesting feature of this model is that the server can be made to execute a generic program, which can cater to different service requests received from a reasonably large number of clients.

Remote procedure calls (RPC) In a remote procedure call, control is transferred from one process to another. The concept of RPC is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single processor. RPC therefore suggests that this same mechanism can be used to transfer control and data across a communication network. Birrell *et al*. [19] provides an implementation of RPC.

In most of the programming models for distributed systems [56], the fundamental concept which differentiates between two models lies in the representation of an inter-process communication. The manner in which an inter-process communication is abstracted forms the foundation upon which a programming model can be built to support specific features that will render the system both efficient and programmable. In the next section, we discuss some of the popular concepts that have been suggested for performing inter-process communications.

2.2.1 Inter-process communications

In message passing parallel programs, the complexity of programming is concentrated in the data communication code segments. In order to simplify expression of these communications during application development, several abstraction concepts for inter-process communications have been suggested (see [6] for an exhaustive survey).

Most of the popular abstraction concepts are implemented as programming language constructs. For example, Hoare's CSP *channels* [62] define communications as send and receive operators '!' and '?' which when applied to a channel results in the transfer of values from the sender to the receiver. Gelernter's LINDA *tuple space* [45], on the other hand, is based on the concept of 'generative communication'. Here, messages are added in tuple-structured form to the computation environment, where they exist as named independent entities until some process chooses to receive them. The distinguishing feature of the tuple space approach is that communications are *orthogonal*. This means that the receiver does not have

prior knowledge of the sender, and the sender does not have prior knowledge of the receiver. EMERALD [20, 87], which is an object-based language, defines *message objects* which encapsulate both static data and an active process. Objects communicate by invoking each other's operations. The distinguishing feature of EMERALD's design is the concept of *object mobility*, which allows a message object to migrate from one processor to another, due to the programmer's intervention or that of the system. Bal *et al.*'s *data objects* in ORCA [13] provides another way of viewing inter-process communications. Here, communications are based on the concept of logically shared data, similar to distributed shared memory systems [5]. The distinguishing feature, however, is that the unit of sharing is a logical, user-defined object rather than a physical, system-defined page. Several advantages of this approach have been discussed in [11]. Yet another programming language is SR [81] which is based on the concept of *capability variables*. Capability variables allow an orthogonal design of the language, which reduces the number of concepts for distributed and parallel programming. Based on this orthogonal design, there are two ways for sending messages (blocking and non-blocking), and two ways for receiving messages (explicit and implicit). A thorough comparison of the above languages—EMERALD, LINDA, ORCA and SR—can be found in [12].

Currently, some of the abstraction concepts are being implemented as application programming interfaces. For example, the Message Passing Interface Forum's MPI Standard [54], and its predecessor, the Parallel Virtual Machine (PVM) [44]. In contrast to providing abstractions as programming language constructs, message passing interfaces provide abstractions for data communications in terms of data structures and library functions. Since the application programming interfaces are closer to the physical systems, and standardised over general programming languages, the application programming interfaces are considerably easier to learn and to deploy, while also able to deliver impressive performance.

In spite of the advantages that low-level inter-process communications have to offer in terms of performance, it has become widely accepted that programming with low-level inter-process communication interfaces must be simplified with higher-level interfaces, so that an application which performs reasonably well can be developed with minimum effort from the programmer. This has resulted in the search for a means of encapsulating low-level details under a suitable abstraction layer. Most of the pragmatic approaches that have been suggested so far are based on the realisation that inter-process communications manifest communication patterns, and that these patterns can be provided as simple programming interfaces.

2.2.2 Skeletal programming models

One of the first models to introduce patterns into parallel programming is the skeletal programming model.

Algorithmic skeletons are defined as higher order functions which correspond to parallel algorithmic structures (or patterns) that occur frequently in parallel programs. The concept was first introduced by Cole [32], in relation to functional programming languages. Cole discussed the implementation of four basic skeletons: (1) divide and conquer skeleton, which can be used for the development of application programs that use algorithms which recursively decompose a problem set into a collection of smaller sub-problems which are further decomposed until they can be solved without further decomposition; (2) iterative combination skeleton, which uses a *greedy* algorithm that tries to combine seemingly uncoordinated sets of objects into a structured combination by applying combination rules within an iteration; (3) cluster skeleton, which uses a reverse abstraction mechanism that defines combination rules based on the peer-to-peer communication pattern presented by systems that arrange processors in a two dimensional grid; and (4) task queue skeleton, which exploits the concurrent progression from problem space to solution space by executing multiple instances of a task, each task execution generating sub-solutions that are added into the task queue, until all the tasks in the task queue have been solved.

Algorithmic skeletons offer several advantages to the programmer:

- The programming effort is reduced because the skeleton implementations encapsulate the best possible low-level parallelisation code which will deliver maximum performance by exploiting the advanced features provided by the underlying system.
- Application programs using algorithmic skeletons are clearer because of the structure defined by the skeletons they use.
- Because of the clearer structure, and encapsulation of the low-level implementation details, the skeleton approach is less prone to programming errors.

As algorithmic skeletons are aimed towards the structured simplification of programming, they have also influenced the development of parallel programming environments based on more traditional approaches, using imperative programming languages—which are arguably more popular than functional programming languages [100]. Advances in imperative skeletal programming environments have been made in two major categories—depending on how the skeletons are implemented.

Compiler based skeleton implementations

Compiler based skeleton implementations define programming languages that are completely new, or extend an existing sequential programming language, in order to support algorithmic skeletons at the compiler level. These languages provide skeletons in the form of programming language constructs (for example, *farm*, *scan*, *reduce*, *map* etc.), which are implemented with low-level, and sometimes architecture dependent code, through a compiler. The compiler based approach offers several advantages:

- The compiler knows the best way to exploit the low-level details of the underlying machine architecture in order to realise a skeleton construct efficiently.
- Since the language does not depend on a host language, it does not suffer from limitations and restrictions that a host language might impose.
- Error checking for incorrect skeleton usage can be performed at compile time; any error that may be detected can also be reported in significant detail—which is sometimes very complicated with a host language.

In spite of the advantages, the compiler based skeleton implementation has the following disadvantages:

- The extension of the supported skeletons changes the definition of the skeleton programming language. Therefore, adding new skeleton constructs breaks the integrity of the language definition, and therefore, reduces the opportunities to attend a concrete compiler implementation.
- If extensions are disallowed, the programmer is again prevented from exploiting new patterns that may arise with the introduction of newer algorithmic structures.
- Arguably, a new language finds it harder to break into the domain of traditional sequential programming languages, and gain acceptance, because more flexible and extensible skeleton implementations can be provided as a library of functions that are implemented on top of sequential programming languages.

Some of the most popular compiler based skeleton implementations are:

P^{3}L The Pisa parallel programming language (P^{3}L) [9, 36, 10] is a well defined programming language that provides skeletons in the form of programming constructs. These skeleton constructs constitute the sole foundation for introducing parallelism to an application program. The P^{3}L language allows skeleton

nesting, which was not supported in the initial implementation of skeletons suggested by Cole [32]. This allows skeletons to contain other skeletons, thus increasing the flexibility and expressiveness of the programming environment. Leopold [71, page 185], however, argues that supporting skeleton nesting introduces several other problems related to the assignment of tasks to the set of processors available for the computation.

SCL The structured coordination language (SCL) [37] provides a larger number of skeleton implementations that are mainly related to data distribution across the processes. This language supports nested data structures which are not supported by the P^3L programming language.

HSM The hierarchical skeleton model (HSM) language is an imperative skeleton language similar to the C [68] programming language. This language focuses on providing *nested data structures* where distribution and alignment of the sequential structures is implicitly defined in the program; reducing the programming effort otherwise required in languages such as the SCL.

HDC The higher order divide-and-conquer (HDC) [59] focuses on the extended implementation of the divide-and-conquer skeleton; so that any algorithm which manifests a divide-and-conquer strategy can exploit the several variants of the skeleton implementation. The base programming language used is a subset of the higher-order functional programming language, Haskell [18]. The HDC, however, uses an eager semantics to enable parallelisation. Important combinators, especially various kinds of divide-and-conquer strategies, are expressed as predefined skeletons. The HDC compiler generates C and MPI parallel target code.

SKIL The skeleton imperative language (SKIL) [21] provides language extensions to the C programming language. These extensions allow higher order functions, and partially support application of functional programming concepts such as *functions* and *polymorphism*. The language introduces the concept of a *parallel abstract data type*, which controls the access of data based on well defined data access patterns, such as the block-oriented access pattern.

Programming library based skeleton implementations

More recently, advances in skeletal programming have been made through library implementations of algorithmic skeletons, such as the Edinburgh skeleton library

(ESKEL) [33, 17], Münster skeleton library (MUESLI) [69], and Fusion-embedded skeleton library [75] which allows skeleton-to-skeleton interfacing. The library approach implements skeletons on top of a host language, and provides the skeletons as a library of functions (or application programming interfaces). This approach has the following advantages:

- The programmer need not learn a whole new language. This reduces the learning curve for programmers who wish to immediately experience the advantages offered by skeleton based models—without leaving the programming language they are comfortable with.
- Easy to extend the set of skeletons provided by the library, as the host language remains the same, and therefore extensions will only mean introducing additional interfaces to the programming library.
- More flexibility to the implementor because the skeleton implementation can use a standard host language which has efficient compiler implementation available for a wider set of architecture, for example the C programming language. The skeletons implemented with such programming languages can then be used on all the architectures supported by the compiler, without re-implementation.

Again, the programming library approach suffers the following disadvantages:

- The skeleton implementation depends on the host language. A skeleton implementor may therefore face certain programming constraints due to restrictions and limitations imposed by the host programming language being used.
- The skeleton implementation cannot take advantage of pattern specific optimisations that can be made by exploiting the machine level details that are only possible with a compiler approach.

Given the advances in compiler technology for a host language, such as the C programming language, we believe, however, that the disadvantages of the programming library approach do not pose a significant problem if we also consider the maintainability of the skeleton implementations.

There exist several other models which promote the same idea of structured parallel programming through frequently occurring pattern abstractions. An exhaustive survey of the *design pattern* based parallel programming models can be found in [76]. Some of the related works are parallel programming ARCHETYPES [29, 74], CO_2P_3S [72, 8], and extensible parallel architectural skeletons [48, 2].

2.2.3 Process group based abstraction models

Most of the abstraction concepts for message passing programming are based on the notion of a 'process group'. We shall now discuss this concept in detail.

A process group is a logical set of processes which is used to derive a collective communication context, normally used to perform communications involving processes in the process group. An exhaustive survey of 'process group' based programming models is given by Chockler *et al.* [30], and articles appearing in the special issue of the *Communication of the ACM*, vol. 38, No. 4, April 1996.

The most important concept which is of significance to this dissertation is the concept of a collective communication. By understanding this, we can understand some of the programming complexities. Since the MPI is the most popular, and most widely available message passing system, we shall now focus our attention on MPI collective communications.

MPI collective communications

Gorlatch suggests that collective communications should be favoured over send-and-receive (or point-to-point) communications [47]. We, however, note that there are cases where send-and-receive primitives are far more effective in terms of both performance and programmability. Consider, for example, algorithms that only require communications between, at most, two processes, for example the Odd-even transposition sorting algorithm (see Section 4.3.3). It is more complicated to implement such algorithms with collective communications, than with send-and-receive primitives. We therefore refine the argument: if the algorithm to be implemented has communication patterns that can be easily realised with collective communications, it is best to use collective communication interfaces. However, if the algorithm can be implemented straightforwardly with send-and-receive primitives, it will be a waste of effort to attempt to fit the algorithm with the available set of collective communications interfaces. In fact, the thesis of this dissertation clearly states our aim: to define a set of interfaces which provides the programmer with the means to express any given communication pattern, without the necessity for transformation of these patterns to fit the interfaces, or the programming model.

In Section 1.1, we introduced the programming issues that arise with process group based abstraction models. We shall now elaborate on this subject, and discuss the implications in more detail. We begin by discussing the concept of a collective communication at its most basic level: the group.

A *group* is defined in the MPI standard [93] as an ordered set of processes where

each process is associated with an integer *rank*. The ranks in a group are contiguously assigned, starting at zero. Groups are represented by opaque data structures and exist locally on a process. They cannot, therefore, be transferred from one process to another. In order to communicate between the participating processes of a group, a communicator should be derived from the group. Unless a communicator is derived, no communication can commence in that group.

A *communicator* is an opaque data structure with a number of attributes, together with simple rules that govern its creation, usage, and destruction. The communicator, in effect, defines a communication domain within which data can be transferred uniquely, and in order. When all the participating processes belong to the same group, the communicator is referred to as an *intracommunicator*. If processes that belong to separate groups communicate, the corresponding communicator is referred to as the *intercommunicator*.

In MPI, any point-to-point or collective communication occurs within a communication domain. Such a communication domain is represented by a set of communicators with consistent values, residing at each of the participating processes: each communicator locally representing the global communication domain on each of the processes on which it is residing. After MPI_Init(), the communicator MPI_COMM_WORLD is created by the runtime system. This communicator is the fundamental communicator from which relevant communicators should be derived. The important points to be noted are:

- The rank of a process depends on the group which is associated with the communicator on which the communication interfaces are invoked. Let us assume therefore that a process has rank r in the communication domain defined by the communicator MPI_COMM_WORLD. Now, if we derive a communicator, say my_comm from MPI_COMM_WORLD, it is not safe to assume that the same process will have rank r in the group that corresponds to the new communicator.

  As the flexibility of the MPI abstraction model derives from the ability to perform communications over boundaries established through communicators [40, pages 295–296], a programmer is meant to define as many communicators as required for a suitable abstraction. This, however, complicates programming because the programmer must keep track of the rank of a process in each of the communicators that have been defined.

- Send and receive calls during a point-to-point communication should specify the same communicator which defines the communication domain of the messages being transferred. This is necessary because the communicator is used to

distinguish between groups of messages.

- A collective communication call involves all of the processes in the group. Most implementations of the collective communications require two barrier synchronisations: one at the start, and one at the end of the call. We will see, in Section 5.4, how this affects the performance of collective communications.
- Collective communications may not use intercommunicators. This follows on from the condition that collective communications should not be invoked on overlapping communication domains.

Programming complexity

The main advantage of collective communications comes from the higher-level abstraction provided by the concept of a communication domain, represented by communicators. Based on this concept, and the conditions for usage discussed in the previous section, participating processes are unified under a common ground of cooperative existence, where a collective communication using the communicator makes it easier to express this cooperation. We can therefore say that once we have a communicator, we are ready to perform communications that have different patterns depending on which collective communication interface is invoked. This also means that, in order to harness these advantages, a programmer must first create suitable communicators. It is therefore reasonable to include the programming costs necessary to derive communicators, while weighing the advantages of collective communications.

Creation of a communicator in the MPI should respect the rationale that:

> "MPI is designed to ensure that communicator constructors always generate consistent communicators that are valid representations of the newly created communication domain ... done by requiring that a new intracommunicator be constructed out of an existing parent communicator ... and this be collective operation over all processes in the group associated with the parent communicator." [93, page 206]

Based on the above rationale, we can extend three arguments. Firstly, a programmer cannot create a new communicator on the fly. Creation therefore means derivation: a new communicator is derived from an existing communicator. Secondly, because the derivation of a new communicator is a collective operation, every call for communicator creation requires two barrier synchronisations, just like collective communication primitives. If this is not preserved, the existence of a globally

```
1  enum { JANUARY := 0, DECEMBER := 11 };
   enum { ACCOUNTANT := 0, RESEARCH, TEACHER, STUDENT, PROJECT };
3  void mpi_artsp_comm ( void ) {
       int world_rank, agrp_rank, rgrp_rank, a_root, r_root, salary[3], month;
5      int a_ranks[] := {ACCOUNTANT, TEACHER, STUDENT};
       int r_ranks[] := {RESEARCH, STUDENT, PROJECT};
7      MPI_Group world, a_grp, r_grp;
       MPI_Comm a_com, r_com;
9      MPI_Comm_rank ( MPI_COMM_WORLD, &world_rank ); /* Get world rank. */
       if ( world_rank = ACCOUNTANT ∨ world_rank = RESEARCH ) {
11         salary[0] := 0; salary[1] := 1000; salary[2] := 2000; /* Set amounts. */
       }
13     /* Get the group associated with MPI_COMM_WORLD. */
       MPI_Comm_group ( MPI_COMM_WORLD, &world );
15     /* For the MPI_Scatter() from ACCOUNTANT. */
       MPI_Group_incl ( world, 3, a_ranks, &a_grp );
17     MPI_Group_rank ( a_grp, &agrp_rank );
       MPI_Group_translate_ranks ( world, 1, &a_ranks[0], a_grp, &a_root );
19     MPI_Comm_create ( MPI_COMM_WORLD, a_grp, &a_com );
       /* For the MPI_Scatter() from RESEARCH. */
21     MPI_Group_incl ( world, 3, r_ranks, &r_grp );
       MPI_Group_rank ( r_grp, &rgrp_rank );
23     MPI_Group_translate_ranks ( world, 1, &r_ranks[0], r_grp, &r_root );
       MPI_Comm_create ( MPI_COMM_WORLD, r_grp, &r_com );
25     /* Start communication. */
       for ( month := JANUARY; month ≤ DECEMBER; month++ ) {
27         if ( agrp_rank ≠ MPI_UNDEFINED ) /* Scatter from ACCOUNTANT. */
               MPI_Scatter ( salary, 1, MPI_INT, salary, 1, MPI_INT, a_root, a_com );
29         if ( world_rank = STUDENT ) salary[1] := salary[0];
           if ( rgrp_rank ≠ MPI_UNDEFINED ) /* Scatter from RESEARCH .*/
31             MPI_Scatter ( salary, 1, MPI_INT, salary, 1, MPI_INT, r_root, r_com );
           if ( world_rank = STUDENT ) salary[0] += salary[1]; /* Sum reduce. */
33         if ( world_rank ≠ ACCOUNTANT ∧ world_rank ≠ RESEARCH )
               printf ( "[%d] My salary: %d\n", world_rank, salary[0] );
35     }
       /* Free the communicators associated with the new scatter groups. */
37     if ( agrp_rank ≠ MPI_UNDEFINED ) MPI_Comm_free ( &a_com );
       if ( rgrp_rank ≠ MPI_UNDEFINED ) MPI_Comm_free ( &r_com );
39     /* Free the scatter groups. */
       MPI_Group_free ( &a_grp ); MPI_Group_free ( &r_grp );
41 }
```

Figure 2.1: MPI implementation of the first decomposition of Example 1.1.1, which uses two scatter collective communications over the groups $\{A, T, S\}$ and $\{R, S, P\}$. A and R are the respective roots of the MPI_Scatter() calls. What is complicated about this implementation is the creation of the communicators for each of the two scatter groups.

invalid communicator is possible. Thirdly, and finally, the derivation of a new communicator from an existing communicator happens in three phases: (1) getting the group that corresponds to the existing communicator. For example, the group that corresponds to the communicator MPI_COMM_WORLD; (2) group inclusion, exclusion etc. so that a subset of the existing group gets selected to form a new group; and (3) create a new communicator from the group just created. In MPI the three operations can be performed by using the interfaces MPI_Comm_group(), MPI_Group_incl() (or MPI_Group_excl()), and MPI_Comm_create() respectively.

Now that we know the process of deriving a communicator, let us see if the MPI approach is simpler when creation of communicators is included in the analysis. In order to perform a critical analysis, let us reconsider the Example 1.1.1, and implement an executable MPI program by using the first decomposition which uses two scatter communications on groups $\{A, T, S\}$ and $\{R, S, P\}$.

The MPI implementation is shown in Figure 2.1. Three groups are declared (line 7): world—which corresponds to the MPI_COMM_WORLD, a_grp and r_grp for the groups that correspond to $\{A, T, S\}$ and $\{R, S, P\}$ respectively. For each MPI_Scatter(), we need a communicator that corresponds to a_grp and r_grp: respectively declared as a_comm and r_comm (line 8). Since a new communicator can only be derived from an existing communicator, we have to first obtain the group, world, that is associated with communicator MPI_COMM_WORLD (line 14). From this group, we create the new groups a_grp and r_grp by using MPI_Group_incl() (lines 16 and 21). Further to the condition that the rank of a process depends on the group, we perform rank translation (lines 17–18 and lines 22–23). Finally, we create the new communicators, a_comm and r_comm from the groups a_grp and r_grp respectively (lines 19 and 24). The MPI_Scatter() calls are invoked within the loop at lines 28 and 31.

In this example implementation, we can observe that the implementation looks pretty simple when we consider only what is happening within the loop at lines 26–35: that is, the usage of the communicators. When the creation of communicators is considered the simplicity, however, vanishes because of the additional concerns involved in properly executing operations related to the management of communicators. For example, at line 14 we derive the process group associated with the communicator MPI_COMM_WORLD. This group consists of all the processes available at the time of initialisation with MPI_Init(). From this group, we derive new groups consisting of only the processes that are necessary to derive the communicators: lines 16 and 21, corresponding to the process groups $\{A, T, S\}$ and $\{R, S, P\}$. As the rank of a process changes with the process group, we derive the new rank of the process from the newly formed groups (lines 17 and 22). We then translate the ranks

of all the other processes in the new group (lines 18 and 23). Finally, we derive the communicators from these newly derived groups (lines 19 and 24). As we can see, although the usage of the communicators is simple and straightforward, deriving the communicators, as required by the condition (see page 21), increases the complexity. We should therefore aim to, (1) simplify the concept of a communicator so that the notion that allows simplification of the abstraction is maintained, though without the additional concerns; (2) further refine the notion of derivation from an existing communicator, so that communicators can be created on the fly, without introducing complications due to faulty communicators, and (3) simplify the association of process ranking with a communicator, or the group, should be simplified by keeping the rank of a process unique and static for the rest of the application, so that the rank becomes an alias to be used for uniquely identifying a process.

In defining a communication domain, it is stated that each communicator is a local representation of a global communication domain. This can be interpreted as the statement that a communicator encapsulates a global state that is visible to all the processes that belong to the corresponding group. This situation is against the principle of distributed systems which states that, in addition to the absence of a common memory and the use of a communication system, distributed systems are characterised by the lack of a global state visible to an observer at any given instant [99, 88]. It is therefore important to explore methods that will allow programmers the benefit of collective communications without breaching this principle. Interpreted in another way, this also demands formation of communicators to be asynchronous, without any barrier synchronisations; which will mean ensuring independence between processes so that every process chooses to complete its task without being delayed by other processes, unless of course required by a data dependency.

2.3 What are the objectives of this dissertation?

The objectives of this dissertation can be summarised as follows:

- Development of an abstraction model which resolves the subtle programming issues related to the 'process group' based models (discussed in Section 1.1). We will focus more on resolving the ambiguity and loss of structural information problem because the other two issues, choice dilemma and performance portability, are resolved as a result. The problem of redundant acknowledgement should also be resolved.

- Development of a programming model which corresponds directly to the abstraction model, so that a programmer can directly translate an abstract representation into a valid application program. It is necessary for the programming model to be non-ambiguous, uniform, expressive and extensible.
- The runtime system which supports the programming model should have an easy way of specifying specialised message buffers. These message buffers should be defined as an integral part of the whole system. The integration should not, however, be too abstract as is the case with LINDA, or too low-level where everything related to the message buffers should be programmed explicitly by the programmer. The aim, therefore, is to provide a reasonable level of abstraction which allows the programmer to specify certain buffer characteristics, but does not force them to program handle creation, management and deallocation of the message buffers.
- Overlapping of computations and communications should be automatic. Since the chances for improving performance by overlapping depend mostly on the runtime execution instance of a program, we consider it necessary to allow the system to take advantage of such opportunities whenever they arise, without explicit programmer intervention.

2.4 How do we plan to attain these objectives?

We plan to achieve the first objective by defining an abstraction model that does not depend on the notion of a 'process group'. In order to avoid usage of the 'process group' concept, we plan to re-analyse the meaning of a communication pattern; and attempt to develop the abstraction model by enhancing the meaning of a sequential control flow graph with our fresh interpretation of a communication pattern. What is integral to this development is an understanding of the differences between sequential and parallel programs; simple send-receive communications and pattern based communications.

Role based parallel programming models such as ACTOR systems [31, 1] have already been suggested. These models define a parallel programming model in terms of agents which participate in a given computation by 'acting' certain roles. In relation to this dissertation, however, the role based model which is highly influential is the concept of a SCRIPT.

Francez and Hailpern [41] introduced the concept of a SCRIPT as an abstraction mechanism which conceals the low-level details that implement patterns of communication. Instead of providing abstractions for point-to-point inter-process com-

munications, SCRIPT aims to provide abstraction for a collection of communications that manifest a communication pattern.

A SCRIPT is defined as a parameterised program section on to which processes enroll in order to participate in a computation. It has three main components: (1) roles, which give the set of instructions that will be executed by any process that enrolls the role, (2) data parameters, which give the set of data variables that are affected by an execution of the role, and (3) body, which is a concurrent program section that defines the patterns of communications defined by the role. For example, in a broadcast, there are transmitter roles which send data to recipient roles, so that data are transferred from the transmitter data parameter to the receiver data parameters, by executing the pattern of communication defined in the body of the SCRIPT, say a spanning tree pattern.

The concept of a SCRIPT has several advantages as an abstraction mechanism. Firstly, it separates the definition of a communication pattern from the executing processes. It is therefore possible to define different SCRIPTs based on recurring patterns, without being influenced by the process environment. Secondly, the concept of enrollment gives a chance to maximise the utilisation of the processing power as processes can enroll on to different SCRIPTs, instead of waiting idle for tasks to be assigned. Thirdly, the communication patterns can be implemented efficiently within the body of the SCRIPT. Finally, the concept of enrollment further allows for a uniform programming interface, which simplifies programming.

In spite of the above advantages, the concept of a SCRIPT poses some issues. Firstly, only one process can enroll on to a given SCRIPT at any time. Therefore, if more than two processes attempt to enroll on to a SCRIPT, all but one process get blocked. If a process is allowed to enroll on to different SCRIPTs at any given moment, there is a possibility for deadlock. Secondly, SCRIPTs work on a fixed network where processes are not created or destroyed dynamically. This means that the execution model is not scalable.

2.4.1 Guidelines from the psychology of programming

To achieve the second objective, we have to understand what factors affect the usability of a programming model. In order to do this, we study results from studies on the psychology of programming. We present some of the interesting results in the following section.

Even though programming languages and programming interfaces are different in the sense of implementation, we will consider them similar on the grounds that they are both tools for expressing the programmer's understanding of the task into

a valid executable program. Hence, on the same ground, we also consider it appropriate to apply arguments related to programming language designs to the analysis and design of programming interfaces.

Green's analysis [52] of programming languages as *Information structures* has emphasised that the structure of information expressed by a programming language should match the structure of the programming task: and depending on this changing structure of tasks, programming language designs should also change in order to reflect the changes required in the information structure. One important programming task is identified as the task of comprehension, of which *deprogramming* is regarded as one very important facet. In deprogramming, after a portion of the mental representation of the problem has been translated into code, it is again translated back to the mental representation as a check. It was found that, in many programming languages, it was easier to develop the code than to recover its meaning. Therefore, by applying this argument to interface design, we argue that a message passing implementation of an algorithm should not be ambiguous about the meaning of the communication pattern involved. This can also be interpreted as: given a communication pattern, the usage of a given set of application programming interfaces should directly result in a single unambiguous implementation of the communication pattern, and this should exactly represent the structure required by the pattern. Additionally, this unambiguous implementation should be guaranteed by the abstraction model, which is what gets translated to the programming interfaces.

Previous research in deprogramming by Pennington [84] and Green [51] has also emphasised the importance of *role expressiveness*, which helps a programmer to identify what the parts of an existing program are, and what is the role or purpose of each part. Since the complications in message passing programs result from the data communication code segments, a programmer should be able to identify which part belongs to data communications, and how those communications actually take place. After applying this argument to interface design, we conclude that interfaces should readily show the parts representing a communication structure. In doing so, the interfaces and related data structures should be designed in a manner so as to assist the programmer in identifying the meaning of a communication pattern as understood from the perspective of each participating process, so that the programmer can comprehend how the processes are interacting as a whole during the communication.

Another study by Pair [82], suggests that there are two aspects to programming: understanding the algorithm and representing objects. The study found out that, in choosing which aspect to consider first, it is better to begin with the algorithm,

and then to choose the necessary representations which will make it possible to efficiently work out the functions and procedures brought to light by the algorithm. The essence of this argument is now a widely accepted paradigm for software engineering, and has significantly influenced practical programming with the introduction of object-oriented programming languages. By applying this argument to interface design, we suggest that structuring of communication patterns into communication objects is inevitable for structured message passing programming. By structuring the communications into such objects, a programmer is able to distinguish and identify, from the various communication patterns, those which are required by a particular algorithm. Subsequent chapters will clarify why this is very important in achieving simplicity, uniformity, and extensibility of the APIs.

Finally, Petre's [85] investigation of the differences in the psychology of language designers and programmers, and Meyer's [79] experience with the design of the Eiffel [78] programming language have concluded that Hoare's guidelines [61, 63],

> "… good language design may be summarized in five catch phrases: simplicity, security, fast translation, efficient object code, and readability …"

are still relevant in the designing of newer programming languages.

From the above discussions, it is quite clear that in the case of parallel programming models, the roles of each process should be easily expressible into a concrete program; and these roles should be easily inferable from the program code so that the previous algorithm on which the implementation is based can be derived from the program without any loss of structural information. Some of the interesting related works are SOFTWARE REFLEXION MODELS due to Murphy *et al.* [80], and the TUBE GRAPH abstraction for reverse engineering due to Mancoridis and Holt [73].

It is also important for the programming model to be simple and straightforward to use once the algorithm has been translated to an abstract model. With regard to simplicity, the two factors that are relevant are uniformity of the programming interfaces, and expressiveness of these interfaces, so that any given algorithm can be directly translated into a concrete representation. If needed, the programming model should also be easily extensible, without the need to modify already existing programming interfaces.

In the next chapter, we shall develop the β-channel abstraction model which will provide us with the concepts that can be implemented into a programming model, which will satisfy the objectives of this dissertation (see Section 2.3).

2.5 Summary

In this chapter, we have put the subject of this dissertation into context (see Section 2.1). We have explored existing and ongoing work on inter-process communications (see Section 2.2.1) and discussed higher-level abstraction models for parallel programming (see Section 2.2.2). We then discussed the abstraction models that are based on the concept of a process group, and illustrated the related programming complexity resulting from its inherent ambiguity (see Section 2.2.3). We then discussed the objectives of this dissertation (see Section 2.3), and derived guidelines by analysing the concepts that have already been introduced in relation to the design of programming languages and systems (see Section 2.4).

CHAPTER 3

Abstraction with communication structures

In the previous chapter, we explored the different approaches that are currently used for message passing programming. In particular, we discussed the different concepts that are defined to introduce patterns into the programming model. In this chapter we develop the β-channel abstraction model which improves the conceptual understanding of a communication pattern, so that the message passing programming interfaces that are provided are non-ambiguous, uniform, expressive and extensible.

We begin this chapter by discussing the practical aspects of a communication pattern, and analysing its meaning in relation to a process which is participating in the communication. More precisely, we ask the question: what does a communication pattern mean to a communicating process? While answering this question, we extend the concept of a *control flow graph*—which is generally used to thoroughly analyse the flow of control in a sequential program—so that the message passing interfaces can be defined as sequential programming primitives. This approach will place the interface invocations closer to a sequential function call, thus simplifying the implementation and usage of a communication pattern. We conclude this chapter by highlighting the advances made towards a clearer understanding of a communication pattern, and outlining the practical advantages it offers to a programmer (treated more thoroughly in the following chapters).

To make future discussions clearer, the new approach will be referred to as the *Communication Structure* approach (or, the β-channel approach).

3.1 Understanding a communication pattern

A message passing program manifests patterns of communication depending on the algorithm it implements. Such patterns define the manner in which the processes interact by passing messages. Algorithms, such as the Mandelbrot set task-farm (see Section 4.3.4), manifest communication patterns that involve the scattering and gathering of data; while others, such as the block-oriented matrix multiplication (see Section 4.3.5), manifest communication patterns which form a ring topology between the processes. A communication pattern, therefore, defines the relationship between processes based on the flow of data across the processes.

As discussed previously (see Chapter 2), higher-level abstraction concepts, such as algorithmic skeletons, are defined to capture such patterns with abstract programming interfaces that help simplification of the programming model. By providing pattern abstractions in the form of programming language constructs, or a library of application programming interfaces, the lower-level implementation details are concealed from the application developer. In general, this approach involving higher-level abstraction offers several advantages (which were discussed previously in relation to specific approaches).

Firstly, the implementation details are hidden under the abstraction layer. Application developers who are not concerned with the pattern implementation can therefore use the patterns without knowing the internal implementation details. Secondly, because the patterns are implemented independent of any specific application, they can be reused in several other applications without re-implementation. Thirdly, the concealment of the implementation details, and the independence from any specific application, gives rise to another advantage: portable efficiency. While implementing the patterns with the lower-level system primitives, proper design decisions can be made in order to harness the potential of a given system architecture, without worrying about the impact that any usage of the pattern might have on the pattern implementation itself. Several implementations, for different system architectures, can therefore be provided for a given pattern without introducing significant differences to the abstraction interfaces. It is therefore expected that any approach for pattern abstraction should provide these advantages to ensure the effectiveness of the approach. We acknowledge, however, that current approaches do not provide all of these advantages to their full potential. Our objective, therefore, is to improve the existing concepts of a communication pattern.

In the next section, we begin development of the β-channel abstraction model by asking the question: what does a communication pattern mean to a process?

Figure 3.1: (a) The meaning of a broadcast when defined using the notion of a process group. All of the processes in the group explicitly invoke the same broadcast interface, but with different parameters. (b) The meaning of a broadcast when defined from the perspective of each process. Each process invokes the interface which best defines the communication it is participating in (in this case, root invokes the interface, s, while receivers invoke r). The realisation of the broadcast pattern is therefore considered as an implied runtime composition of the process specific interface invocations.

What does a communication pattern mean to a process?

Given a set of communicating processes, we have defined a communication pattern as the manner in which data flows across this set of processes. This definition, however, only provides a superficial idea of what a pattern really is: which is, the way in which all the processes interact together during the communication. If each process is observed as an independent entity—which they all are if we focus on the execution of one process at a time—does the conceptual meaning of the pattern change with the process?

To answer this question, let us consider a simple pattern which is usually provided in most approaches for pattern abstraction: a message broadcast. A *message broadcast* is the sending of a message to all the members of a group of processes so that all of the member processes receive the same message [101].[1]

The direct approach is defining the pattern on the group, as shown in Figure 3.1.a. Here, all of the processes participating in the communication invoke a broadcast interface. How each process behaves once the interface is executed is decided based on the parameters being passed during the invocation (e.g. the receiving processes define the root parameter, while the sender process ignores this parameter). In this approach, the pattern is therefore explicitly defined by the interface which all of the processes invoke. This, we have observed previously (see page 4), results in the loss of structural information.

Another realisation of the broadcast pattern, due to the β-channel approach, is

[1] Although there are different ways in which the broadcast pattern can be implemented [90, 97], at this point we will only focus on the abstract meaning of the pattern.

shown in Figure 3.1.b. Here, the pattern is defined not as a single interface which is invoked by all the processes, but rather a set of interfaces that are invoked by all the processes based on their interpretation of the broadcast pattern. The root process sends data to the receiving processes, therefore invoking an interface s which sends the message to all the receiving processes. The receiver processes, on the other hand, invoke the interface r for receiving the message that is sent by the root process. The broadcast pattern is, hence, not associated explicitly with the group, but is considered to be an implicit runtime composition of the process specific interface invocations.

One may ask how, then, is the pattern abstraction concretely implemented in the application program when it is only defined implicitly? The answer to this question lies with the interface s, which the root invokes. If we were to use point-to-point interfaces instead of invoking a single interface s, then no instance of pattern abstraction is being used. This is because such a method would mean sending data to all the receiver processes one by one—which in itself does not readily show the broadcast pattern. However, if we encapsulate these communications within an appropriate interface, say interface s, then we have achieved pattern abstraction because the internal details concerning the sending of messages to all the receiver processes one by one are concealed by s. If we observe carefully, this makes sense because the root is the only process which actually require a broadcast pattern; none of the receiver processes requires a broadcast pattern because all they are required to perform is the acceptance of data from the root—which in principle is point-to-point communication, entirely different from a broadcast.

From this discussion, we can observe that the meaning of a communication pattern does change with the process under consideration. We therefore suggest that it is more meaningful to associate localised communication patterns with the participating processes, while leaving the holistic pattern as an implied runtime composition of these process specific patterns; rather than explicitly defining the holistic pattern as an interface which is then invoked by all of the participating processes. To demonstrate one major advantage of the β-channel approach, we will now resolve the *ambiguity*, and *loss of structural information* problem (see page 4).

Resolving the ambiguity and loss of structural information problem

In Section 1.1, we discussed the ambiguity problem related to the overlapping of communication domains when process group based abstraction models are used. To resolve this problem, let us recall Example 1.1.1 which introduced the problem.

The ambiguity problem occurs when a single collective communication cannot

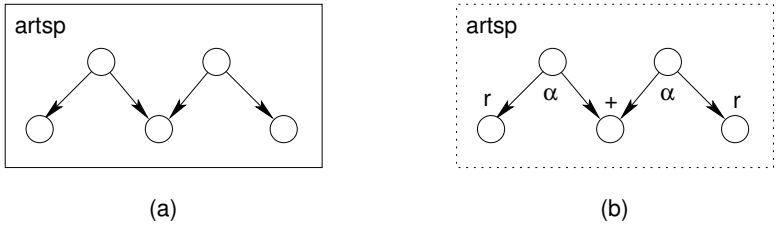


Figure 3.2: (a) A process group based collective communication interface for the pattern in Example 1.1.1. (b) The same pattern implemented with interfaces based on the process specific interpretations of the pattern.

be invoked over a group of processes because the communication domains defined by the process groups overlap. Although one may argue that a collective communication, say *artsp*, which encapsulates the pattern of Example 1.1.1, can be defined on the group $\{A, R, T, S, P\}$ (Figure 3.2.a), this approach is beset by several pragmatic concerns. Firstly, how does one manage the patterns if new patterns are defined for every communication pattern that may appear in a set of applications? Secondly, the approach contradicts the previously agreed condition that pattern abstractions should be independent of any particular application. Thirdly, a pattern implementor cannot design a pattern unless the application that will use the pattern is specified. This means that the application programmer should also be the pattern implementor. Finally, these arguments also mean that re-usability and performance portability cannot be ensured in such approaches.

The only solution, therefore, is to decompose the pattern into non-overlapping communication domains, which can then be used to invoke the respective collective communication interfaces that are properly designed and implemented by a pattern implementor. We have, however, noted already that there is no straightforward method for performing this decomposition, as it can be done in different ways, which result in different implementations of the same pattern: hence the ambiguity.

On the other hand, if we consider the β-channel approach, which implicitly defines a holistic pattern based on the process specific localised patterns, the pattern in Example 1.1.1 is realised as shown in Figure 3.2.b. Here, the holistic communication pattern is implemented by invoking the following interfaces: A and R invoke scatter interface (α), S invokes data reduction interface ($+$), and T and P invoke receive interface r. Since these interfaces (and their associated localised patterns) are specific to the process and independent of the other processes in the group, there can exist only one implementation for a given pattern, and that implementation precisely defines what the pattern actually means, without any loss of structural information.

3.2 A sequential foundation: the control flow graph

Every process in a message passing program is in itself a sequential process: executing a set of instructions sequentially. This set of instructions includes the message passing interfaces that can be considered as sequential instructions if we conceal the data dependency which relates the interface to corresponding interfaces on other processes. To define an abstraction model for message passing programming, we therefore need a clear understanding of what each process does during a parallel execution. In order to obtain this, we revisit the fundamental concepts that are defined by the control flow graph of a sequential program.

A control flow graph defines the flow of control throughout the set of instructions in a program. It is best represented by a directed graph, defined as follows:

Definition 3.2.1 (**Directed graph**)
A *directed graph* G is denoted by $G = (N, E)$ where N is the set of nodes (or vertices) $\{n_1, n_2, \ldots\}$ and E is the set of directed edges (or arcs) $\{(n_i, n_j), (n_j, n_k), \ldots\}$. Each directed edge in the graph G is represented by an ordered pair of nodes (n_i, n_j), where n_i and n_j are not necessarily distinct. [4] ▮

In the directed graph representation of a control flow graph, each node represents a *basic block*: defined as a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed) [4]. A block can have predecessor or successor blocks when it does not represent the beginning or end of the program. During iterations and recursions, it can be a successor of its own. Program entry blocks might not have predecessors that are in the program; program terminating blocks never have successor blocks in the program.

The edges of the directed graph, on the other hand, represent the flow of control from one basic block to the other. This defines the order of execution of instructions that are in different blocks.

To provide a more high-level abstraction of the control flow graph, subsets of the control flow graph with basic blocks are encapsulated within an *extended basic block*. This defines a sequence of program instructions each of which, with the exception of the first instructions, has one and only one immediate predecessor and that predecessor precedes it, though not necessarily immediately [3]. Extended basic blocks can be formed from the tree of basic blocks resulting from programming constructs such as the if...then...else clause.

The extended basic block allows grouping of basic blocks, and therefore allows analysis of a program at different levels of detail. We will refer to this level of

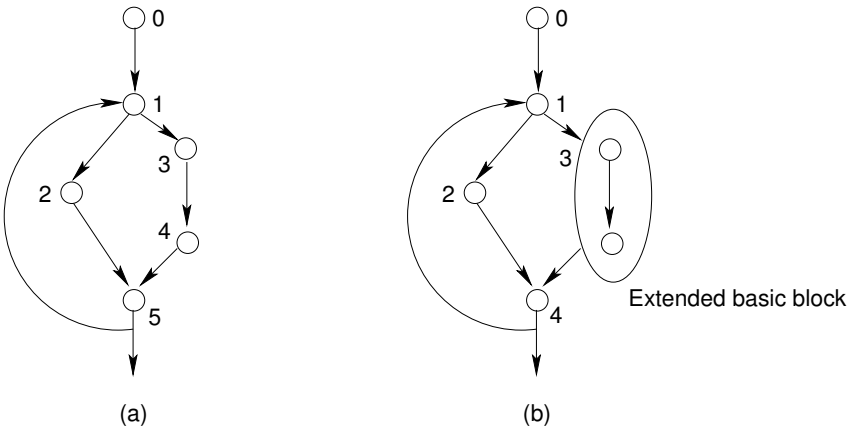


Figure 3.3: The resolution of (a) is higher than the resolution of (b) because (b) encapsulates the basic blocks, nodes 3 and 4, into a single extended basic block, node 3.

detail as the *resolution* of the nodes in the control flow graph. For example, the control flow graph shown in Figure 3.3.a has higher resolution than the one shown in Figure 3.3.b, because the control flow graph shown in Figure 3.3.b encapsulates two basic blocks, nodes 3 and 4, with a single extended basic block, node 3.

Following the previous discussion, an extended basic block can therefore include invocations to interfaces that are related to message passing, if we conceal the data dependency between the local nodes and the nodes existing on remote processes. To account for the data dependency when representing message passing parallel programs with control flow graphs, we introduce the concept of a dependency point.

From the next section onwards, we will introduce new concepts which extends the notion of a control flow graph so that communication patterns can be integrated within the message passing interfaces. Although some of these concepts may be related to existing ones, it is defined explicitly to avoid confusion.

3.3 Towards parallelisation: the dependency point

In a message passing parallel program, processes share data: producer processes produce data that is used by consumer processes. Data are transferred from the producer to the consumer by transmitting a message containing the data through a communication channel (or a link), shown in Figure 3.4. This channel has two opposite ends, the *source* and the *sink*. A producer sends a message by putting the message into a sink, while a receiver receives a message by retrieving the message from a source.

The manner in which a process behaves as a producer or consumer cannot be defined for the whole program because every process can behave both as a producer

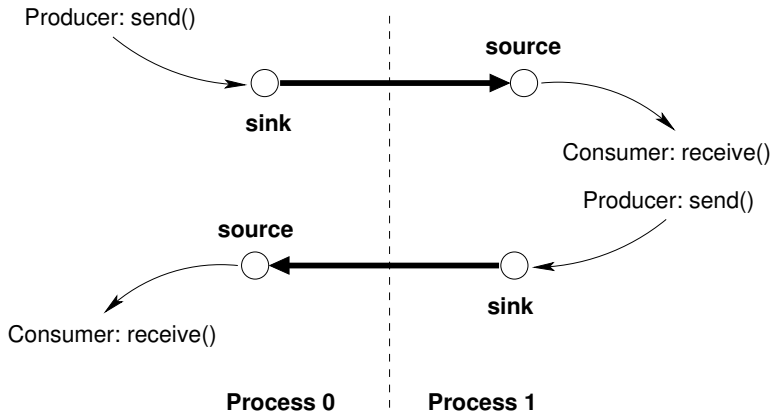


Figure 3.4: The definition of a *source* and a *sink* from the perspective of the invoking process, when the process behaves either as a producer or a consumer. A producer sends a message by putting the message into a sink, while a consumer receives a message by retrieving the message from a source. The source and sink are opposite sides of a communication channel (or a link).

and a consumer depending on the instruction it is executing at a certain moment, shown by the interchanging producer and consumer behaviour in Figure 3.4.[2] It is therefore necessary to associate the behaviour (producer or consumer) of a process at the instruction level, depending on whether the instruction is used to send or receive data.

To differentiate the extended basic blocks that have message passing instructions from the ones that only use locally available data, the nodes with message passing instructions in the control flow graph are defined as follows:

Definition 3.3.1 (**Dependency point**)
A *dependency point* is a node on the control flow graph that encapsulates instructions with data dependencies spanning outside the process, to other processes available during the computation. ▌

A dependency point is further classified as follows:

Definition 3.3.2 (**Sink dependency point**)
If the instruction that is executed within a dependency point results in the transfer of data from the local address space to a sink, the dependency point is referred to as the *sink dependency point*. ▌

[2]For clarity, we use 'instructions' instead of 'interfaces'. This is to put the discussion more in context with those related to a control flow graph.

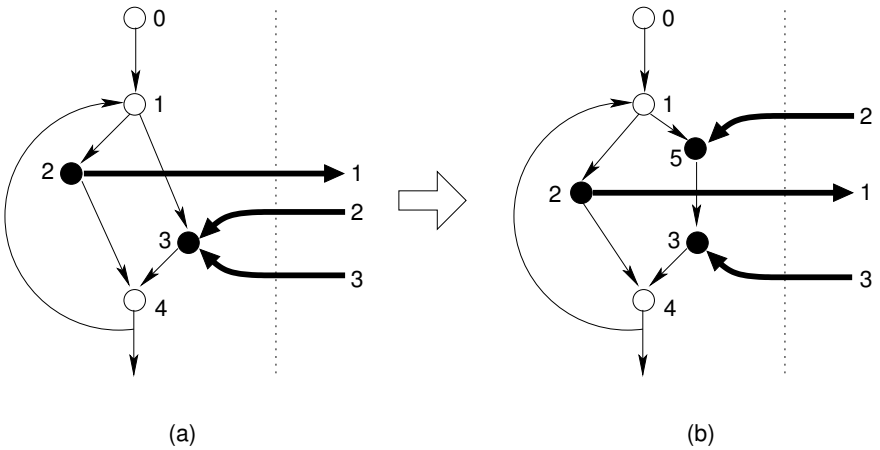


Figure 3.5: (a) The control flow graph has five extended basic blocks represented by the nodes 0, 1, 2, 3, and 4; of which node 2 is a valid sink dependency point, while node 3 is an invalid dependency point. (b) Node 3 is transformed into two valid source dependency points, node 3 and node 5, by increasing the resolution of the invalid node through decomposition.

Definition 3.3.3 (**Source dependency point**)
If the instruction that is executed within a dependency point results in the retrieval and transfer of data from a source to the local address space, the dependency point is referred to as the *source dependency point*. ▌

In order to make the above classifications effective, the resolution of each node should be such that it contains only one instruction that interacts with remote processes by either receiving or sending data. We define the validity of a dependency point as follows:

Definition 3.3.4 (**Validity of a dependency point**)
A dependency point is said to be valid if it encapsulates exactly one instruction for interacting with a remote process: either as a producer or a consumer. ▌

In Figure 3.5.a, for example, node 2 is a valid sink dependency point. Node 3, on the other hand, is not a valid source dependency point because it encapsulates two off-process data accesses. To make node 3 valid, it should be broken down such that each of the resulting nodes accesses a single off-process data only (thus increasing the node resolution). This is shown in Figure 3.5.b where what is originally node 3 is broken down into node 3 and node 5; each accessing separate off-process data.

To identify the dependency points that belong to a process, a collection of dependency points on a process is defined as follows:

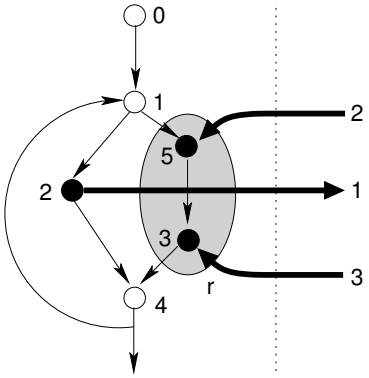


Figure 3.6: In order to define a communication pattern which encapsulates the point-to-point communication represented by the dependency points, nodes 3 and 5, a dependency class r is defined as a logical group of the two dependency points.

Definition 3.3.5 (**Dependency set**)
For a process, if the sets Γ^- and Γ^+ respectively represent the set of all the sink and source dependency points in the control flow graph, a *dependency set* Γ for the process is defined as the union $\Gamma^+ \cup \Gamma^-$. That is, $\Gamma = \Gamma^+ \cup \Gamma^-$. ▌

A dependency point is represented by a node that has an incoming or outgoing edge (shown in Figure 3.5 with thick arrows), the other end (head or tail) of which is incident on a node that does not belong in the dependency set of the process under consideration. Such edges are referred to as *dependency edges*, and are defined as follows:

Definition 3.3.6 (**Dependency edge**)
A *dependency edge* $e = (a, b)$ is an ordered pair of nodes where either $a \in \Gamma$ and $b \notin \Gamma$, in which case it is referred to as a *sink dependency edge*; or $a \notin \Gamma$ and $b \in \Gamma$, in which case referred to as a *source dependency edge*. The classification of a dependency edge into source or sink dependency edge is relative to the process which defines the set Γ—one process's sink is another process's source, and *vice versa*. ▌

The *order of dependency* for a dependency edge is given by the directed edge so that the tail of the edge is incident on the sink dependency point, while the head is incident on the source dependency point. Both dependency points incident on the dependency edge belong to different processes. In essence, the order of dependency gives the direction of data flow from the producer sink to the consumer source.

3.4 Towards pattern abstraction: the dependency class

We have discussed in Section 3.3 how a dependency point represents one end of a point-to-point communication. In this section we discuss how a localised pattern is defined on the dependency set of a process.

Given a set of point-to-point communications, pattern abstraction mechanisms group these communications based on a communication pattern so that such groups can be represented by a single interface invocation which conceals the underlying point-to-point communications. In the β-channel approach, therefore, we need a mechanism to group the dependency points that belong to the dependency set of a process, so that a pattern can be defined on the group. The β-channel approach defines such logical groups as follows:

Definition 3.4.1 (**Dependency class**)
A *dependency class* is an equivalence class on the set of dependency points that are incident on a path originating from the root of the control flow graph such that it does not contain nodes other than source or sink dependence points. ▌

A dependency class represents a logical group to which any action (send or receive) that is applied gets translated to the internal invocation of the underlying point-to-point communications. The communications and related computations are performed by the runtime system in such a way that the communication pattern associated with the dependency class is properly realised during execution. As an example, assume that the data received at the source dependency points 3 and 5 in Figure 3.5.a should be added to give a sum reduced value. To define a sum reduction pattern on these two dependency points, we group them under a dependency class, r (see Figure 3.6).

The *degree* of a dependency class r is defined as the number of dependency points within the class. It is denoted by $\delta(r)$. The *in-degree* of a class r is defined as the number of source dependency points within the class r, and is denoted by $\delta^+(r)$. The *out-degree* of a class r is defined as the number of sink dependency points within the class r, and is denoted by $\delta^-(r)$. For any given class r, $\delta(r) = \delta^+(r) + \delta^-(r)$.

Before we define a pattern on a dependency class, let us first observe some conditions that should be satisfied in order to ensure that the pattern definitions are consistent with the set of actions applicable to the dependency class.

Definition 3.4.2 (**Validity of a dependency class**)
A dependency class r defined on a process with dependency set Γ is said to be valid if and only if,

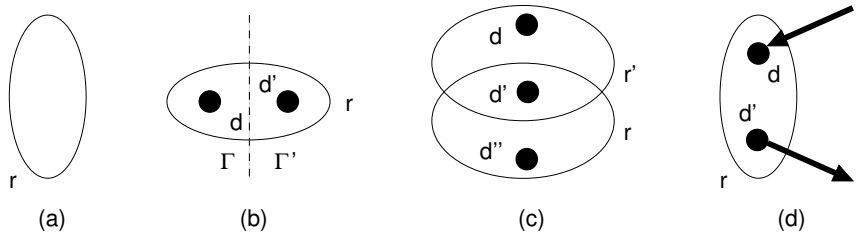


Figure 3.7: Invalid dependency classes: (a) $r = \emptyset$, (b) $d \in \Gamma$ and $d' \in \Gamma'$, (c) $r \cap r' \neq \emptyset$, and (d) $\delta^-(r)$ and $\delta^+(r)$ are both 1. Here r and r' represent dependency classes, Γ and Γ' represent dependency sets on different processes, d, d', and d'' represent dependency points.

1. $r \neq \emptyset$,
2. $d \in r \Rightarrow d \in \Gamma$,
3. $d \in r_i \Rightarrow d \notin r_k$, for $i \neq k$ assuming r_i and r_k are defined,
4. $\delta^+(r) > 0 \Rightarrow \delta^-(r) = 0$, and *vice versa*.

Here d is a dependency point; r_i and r_k are different dependency classes. ∎

The first condition requires that every dependency class should have at least one dependency point. If the dependency class r is empty then no communication pattern can be defined because there are no dependency points to be activated when an action is applied to r. The second condition requires that every dependency point within a dependency class should also be a member of the dependency set defined on the process under consideration. This means that dependency classes should only group local dependency points, and therefore should not encapsulate dependency points defined on other processes. The third condition establishes mutual exclusion of the dependency classes defined on a process, so that actions applied to one dependency class do not interfere with other classes. The last condition ensures that all the dependency points within a class represent either the tail or the head of the dependency edge so that any one (but not both) of the actions, send or receive, is defined on the dependency class.

The dependency classes in Figure 3.7 are invalid: (a) condition 1 is not satisfied because there are no points of dependency, (b) condition 2 is not satisfied because class r encapsulates dependency points that exist on different processes, (c) condition 3 is not satisfied because $r \cap r' \neq \emptyset$, and (d) condition 4 is not satisfied because $\delta^-(r)$ and $\delta^+(r)$ are both 1.

Before proceeding further with the definition of communication patterns, let us first describe what we mean by applying an action to a dependency class, so that the following sections on pattern abstraction make sense. We will refer to the

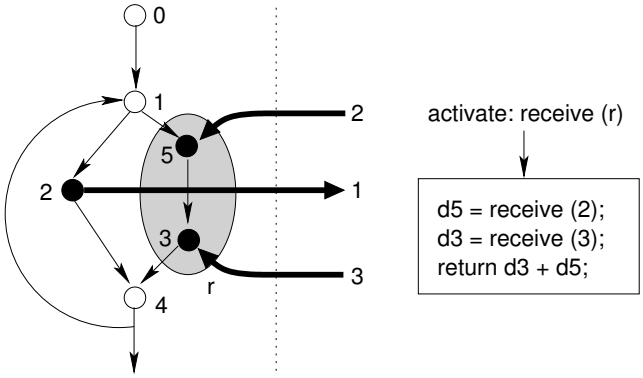


Figure 3.8: Activation of the dependency class r (see Figure 3.6) for receiving data. The right-hand side shows the internal instructions that are executed.

application of an action to a dependency class as the activation of the dependency class, which is described in the next section.

3.5 Initiating a communication: the dependency class activation

The *activation* of a dependency class r is defined as an execution event where all of the dependency points within r are activated in order to communicate data. As shown in Figure 3.8, for example, a dependency class which consists of the dependency points 3 and 5 is said to be activated when a single receive instruction is executed on r. This activation eventually results in the execution of the separate receive instructions at each dependency point, following which the received data are reduced to give their sum.

In Section 3.3, we emphasised that the manner in which a process behaves either as a producer or a consumer cannot be defined for the whole program that the process is executing. It was suggested therefore that the producer or consumer behaviour is best associated at the instruction level. Furthermore, from the definition of a dependency class, we know that activating a dependency class results in the internal invocation of the encapsulated point-to-point communications; and with the necessary condition 4 from the definition of a valid dependency class (see Definition 3.4.2) , it is ensured that an activation is semantically valid—giving a correct representation of the point-to-point communications if the abstraction provided by the dependency class is ignored.

When a process behaves as a producer, it sends data. In the programming model,[3] this action is represented by the interface bc_put(). This interface is invoked

[3]From this point onwards, we will briefly describe the programming interfaces in conjunction with

by passing the data structure which represents the dependency class, the application buffer to send the data from, and the number of data units that should be sent.

When a process behaves as a consumer, it receives data by invoking the interface bc_get(). This interface is invoked by passing the data structure which represents the dependency class, the application buffer where the received data should be stored, and the number of data units that should be received.

Due to the fact that the communication pattern is defined within the dependency class, these two interfaces, bc_put() and bc_get(), are sufficient to initiate any form of communication. This is quite contrary to the MPI approach where the communication pattern is associated with the interfaces, for example MPI_Scatter() for scattering data, MPI_Gather() for gathering data, and so on.

We now proceed to the definition of a communication pattern. In the next section, we provide answers to the following questions: how are all the point-to-point communications within a dependency class executed to manifest a pattern of communication? What happens when either bc_put() or bc_get() is invoked by a process?

3.6 Defining communication patterns: the role

When a dependency class is activated, the dependency points within that class can in turn be activated internally in different ways. For example, finding the maximum value of all the data that has been received, or summation of all the values that have been received. By keeping the degree of the class constant, different internal events can be defined on the dependency points so that activation of the same dependency class results in different meanings. These internal events, in fact, defines the local communication pattern for the process activating the class.

In the β-channel approach, holistic patterns are implicitly defined as the composition of localised patterns. These localised patterns are in turn defined in a dependency class by associating with it a semantic property which relates all the dependency points within that class. We define this property as the *role* of that process in that dependency class.

Definition 3.6.1 (**Role**)
The *role of a dependency class*, which in turn represents the role of the process in that dependency class, is defined as the pattern of internal events that are executed on the dependency points within r, when r is activated. ∎

---

the conceptual definitions whenever it is appropriate. These interfaces will be discussed in detail once all of the relevant concepts have been introduced. Note that all the interface names are prefixed with 'bc_', which stands for 'branching channel'.

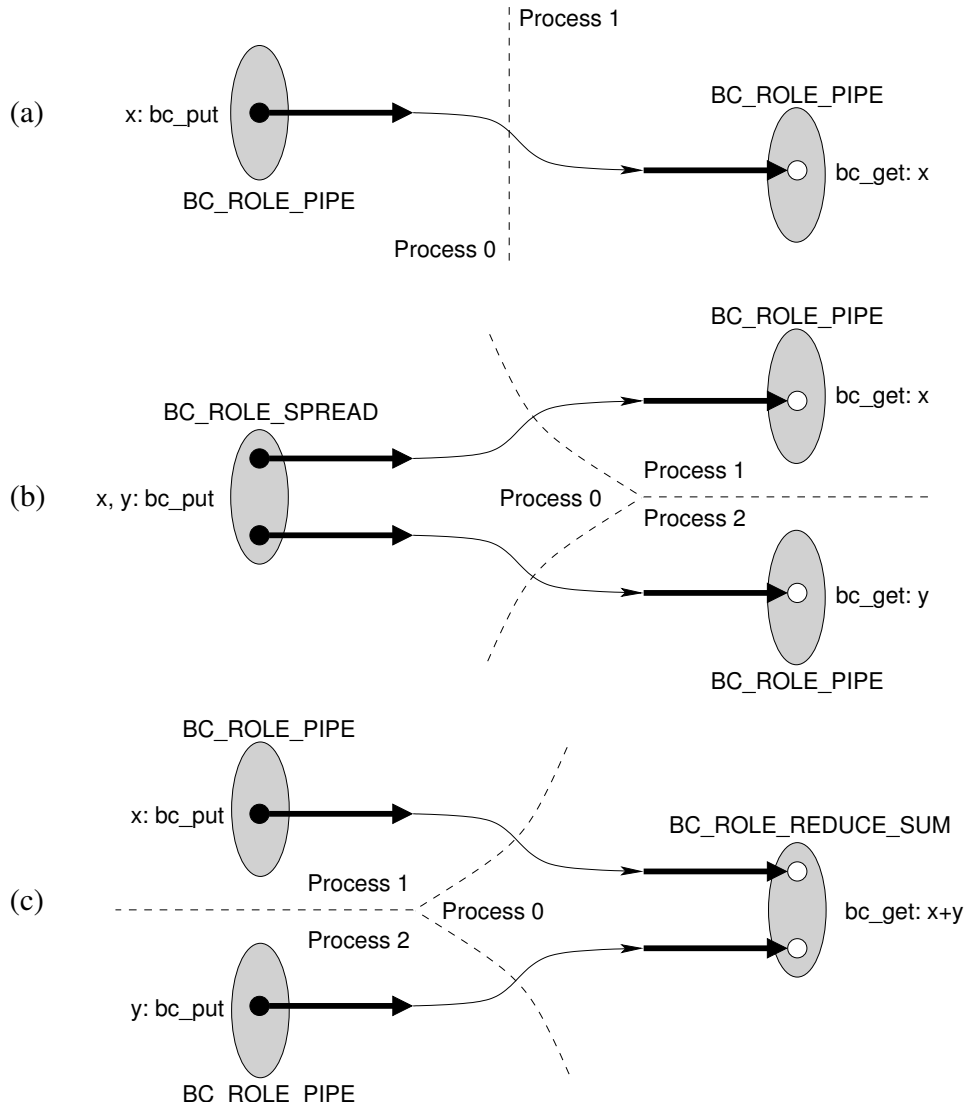


Figure 3.9: Semantics of the dependency class roles, BC_ROLE_PIPE, BC_ROLE_SPREAD, and BC_ROLE_REDUCE_SUM. (a) BC_ROLE_PIPE role can be used in dependency classes with both source and sink dependency points. Through the dependency edge created by the two dependency points, data x is sent from the application buffer on $\mathcal{P}_0$ and received in the application buffer on $\mathcal{P}_1$. (b) BC_ROLE_SPREAD role can only be associated with dependency classes having only sink dependency points. Data x and y on $\mathcal{P}_0$ are sent from the application buffer so that $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively receive x and y, but not both. (c) BC_ROLE_REDUCE_SUM role can only be associated with dependency classes having only source dependency points. Here, data x and y from $\mathcal{P}_1$ and $\mathcal{P}_2$ are received on a temporary buffer on $\mathcal{P}_0$, and their sum is finally stored in the application buffer.

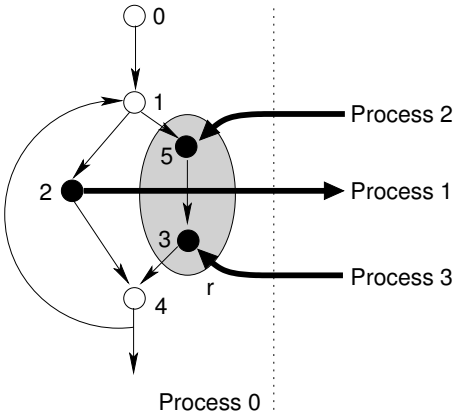
Although all of the implemented roles are discussed thoroughly in Section 4.2, we will briefly discuss here three roles which we will use for a demonstrative implementation of Example 1.1.1. The roles that we will discuss here are BC_ROLE_PIPE, BC_ROLE_SPREAD, and BC_ROLE_REDUCE_SUM. These roles correspond to the localised communication patterns required by Example 1.1.1. The BC_ROLE_PIPE role has two meanings depending on the type of dependency point. In the first case, if r is a dependency class with a single sink dependency point, which is associated with a BC_ROLE_PIPE role, activation of this dependency class with bc_put() results in the sending of data from the buffer to the dependency edge incident on the dependency point. In the second case, if r is a dependency class with a single source dependency point, which is associated with a BC_ROLE_PIPE role, activation of this dependency class with bc_get() results in the retrieval and transfer of data from the dependency edge incident on the dependency point to the application buffer.

The role BC_ROLE_SPREAD, on the other hand, can only be associated with a dependency class with sink dependency points. This means that this role can only be used for defining producer patterns. If r is a dependency class with n sink dependency points, and if this is associated with a BC_ROLE_SPREAD role, activation with bc_put() results in the transfer of *unique* data from the buffer to the dependency edges incident on the sink dependency points.

Similar to the BC_ROLE_SPREAD role, the BC_ROLE_REDUCE_SUM role can only be associated with a dependency class with source dependency points. This means that this role can only be used for defining consumer patterns. If r is a dependency class with n source dependency points, and if this is associated with a BC_ROLE_REDUCE_SUM role, activation with bc_get() results in the retrieval of n data units from the dependency edges incident on the source dependency points, and storage of the sum of the data hence received in the application buffer. In Figure 3.9, we show graphical representations of the semantics of these roles.

3.7 Putting it all together: the communication structure

In the previous sections, we have defined the data dependency points which give the nodes of the control flow graph where the process interacts with a remote process while sending or receiving data. We have also defined a dependency class as a set of dependency points satisfying a certain set of conditions (see Definition 3.4.2). To define the localised pattern which specifies the manner in which all of the dependency points within a dependency class are activated, we introduced the concept of a role.



(a)

Dependency point	Projection (λ)
5	P_2
2	P_1
3	P_3
Dependency class	**Projection (κ)**
r	$\{P_2, P_3\}$
Dependency class	**Domain (τ)**
r	$\{P_0\}$

(b)

Figure 3.10: (a) The source dependency point 2 and sink dependency points 5 and 3. (b) The projection of the dependency points 2, 3 and 5, the projection of the dependency class r with dependency points 3 and 5, and the domain of the dependency class r. P_i represents process with rank i.

In this section, we define a *communication structure*: the concept which incorporates all of the other concepts to finally define the holistic communication pattern of an application program.

Definition 3.7.1 (**Communication structure**)
A *communication structure of a computation* involving k processes is defined as a k-partite simple directed graph $G = (D, E, R)$ where $D = \cup_{i=0}^{k} \Gamma_i$ is the set of all the dependency points defined on all the participating processes. The set E is defined as the set of directed edges that is incident on a pair of source and sink dependency points that exist on different partitions. The set R is defined as the set of equivalence classes that are defined on each of the k processes. ∎

The communication structure represents the network of links between all of the processes participating in the computation. This network represents the manner in which messages are communicated between the processes. The structure of this network gives the holistic communication pattern manifested by the implemented algorithm. Before we discuss the validity of a communication structure, let us first define the following concepts.

For a given dependency class, we are interested in knowing the partition in which it is defined. We refer to this as the domain of the dependency class.

Definition 3.7.2 (**Domain of a dependency class**)
The *domain of a dependency class*, $\tau(r)$, for a dependency class, r, is defined as the

partition on which all of the dependency points within r are defined. ▌

For any given dependency point incident on a dependency edge, we are sometimes interested in the partition on which the other dependency point is defined. We refer to this as the projection of the dependency point, defined as follows:

Definition 3.7.3 (**Projection of a dependency point**)
The *projection of a dependency point*, $\lambda(x)$, for a dependency point x is defined as the partition on which the other vertex of the directed edge on which x is incident is defined. ▌

For a sink dependency point, the projection gives the rank of the receiver process; for a source dependency point, the rank of the sender process.

While dealing with dependency classes, we are also interested in knowing the projections of all the dependency points within the dependency class. We refer to this as the projection of the dependency class, defined as follows:

Definition 3.7.4 (**Projection of a dependency class**)
The *projection of a dependency class*, $\kappa(r)$, for a dependency class r is defined as the 'unordered' set of the projections of all the dependency points in r. Mathematically, $\kappa(r) = \{\lambda(x) \mid x \in r\}$. ▌

For example, in Figure 3.10, the projections of the dependency points 5, 2 and 3 are 2, 1 and 3 respectively. The projection of the dependency class r with dependency points 5 and 3 is the unordered set $\{2,3\}$.

In the programming model, the projection of the dependency class is represented with a data structure referred to as the *process list*. The process list is a process specific data structure which encapsulates the projection of a dependency class, but specialises it with 'ordering' information. When a role is associated with a dependency class, the semantics of that role define the pattern in which the dependency points are coordinated. For some roles, such as the BC_ROLE_SCATTER, the coordination requires ordering of the dependency points in order to specify how the data from the sender application buffer is sent. For other roles, for example BC_ROLE_REDUCE_SUM, such ordering is not necessary, and therefore it does not require ordering information in addition to the projection of the class. This is the reason why we ignore ordering information while defining the projection of a dependency class.

We now define the validity of a communication structure as follows:

Definition 3.7.5 (**Validity of a communication structure**)
A *communication structure* $G = (D, E, R)$ is said to be valid if and only if,

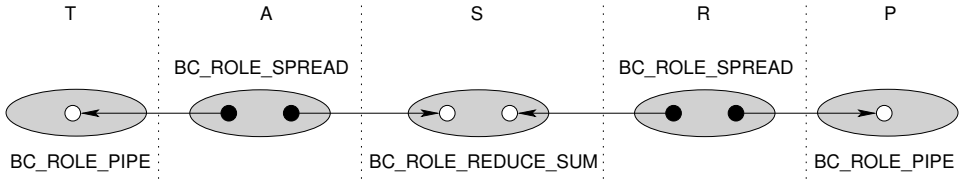


Figure 3.11: The communication structure for the communication pattern of Example 1.1.1. Each process creates a dependency class: T creates a source dependency class with one dependency point whose projection is $\{A\}$. It has the role BC_ROLE_PIPE. We can similarly derive the information for P. Both A and R, on the other hand, create a dependency class with two dependency points, with projections $\{T, S\}$ and $\{S, P\}$ respectively. Both have BC_ROLE_SPREAD role. Finally, S creates a dependency class with two dependency points, with projection $\{A, R\}$. It has the BC_ROLE_REDUCE_SUM role.

1. $\forall r \in R$, r is a valid dependency class (see Definition 3.4.2),
2. $\forall r \in R$, $\kappa(r)$ contains unique members,
3. $\forall r \in R$, $\kappa(r) \cap \tau(r) = \emptyset$, and
4. $\sum_{i=0}^{|R|} \delta(r_i) = 0$ for all $r_i \in R$.

∎

The first condition ensures that all the dependency classes in R can be activated with either bc_put() or bc_get(). The second condition ensures that any dependency class $r \in R$ with degree $\delta(r) > 1$ does not have more than one dependency edge that is incident on the same remote partition. This is necessary because receiving more than one data unit with separate receive calls can be combined into a single receive call that has multiple data units. The third condition ensures that all the directed edges cross the partition boundary, so that every dependency edge is incident on two dependency points defined in different partitions. In practice, this means that a dependency edge can only be used for transferring data from one process to another, and therefore cannot be used for transferring data within the process. In fact, communicating data to self does not make sense because the process already has access to the data through a direct memory access. The final condition ensures that all the dependency classes are connected to the required number of dependency classes on other partitions so that every source is connected to its respective sinks, and *vice versa*.

3.8 The encapsulating data structure: the branching channel

While developing an application, the communication structure is created as a runtime composition of process specific data structures referred to as the branching

```
1  enum { JANUARY := 0, DECEMBER := 11 };
   enum { ACCOUNTANT := 0, RESEARCH, TEACHER, STUDENT, PROJECT };
3  void artsp_branching ( void ) {
     bc_chan_t *src := NULL, *sink := NULL;
5    bc_plist_t *src_pl := NULL, *sink_pl := NULL;
     int salary[2], month;
7    if ( bc_rank = ACCOUNTANT ∨ bc_rank = RESEARCH ) {
       salary[0] := 1000; salary[1] := 2000; /* Set amounts. */
9    }
     /* Create communication structure. */
11   switch ( bc_rank ) {
     case ACCOUNTANT:
13     sink_pl := bc_plist_create ( 2, TEACHER, STUDENT );
       sink := bc_sink_create ( sink_pl, bc_int, 1, BC_ROLE_SPREAD );
15     break;
     case RESEARCH:
17     sink_pl := bc_plist_create ( 2, STUDENT, PROJECT );
       sink := bc_sink_create ( sink_pl, bc_int, 1, BC_ROLE_SPREAD );
19     break;
     case TEACHER:
21     src_pl := bc_plist_create ( 1, ACCOUNTANT );
       src := bc_src_create ( src_pl, bc_int, BC_ROLE_PIPE );
23     break;
     case STUDENT:
25     src_pl := bc_plist_create ( 2, ACCOUNTANT, RESEARCH);
       src := bc_src_create ( src_pl, bc_int, BC_ROLE_REDUCE_SUM );
27     break;
     case PROJECT:
29     src_pl := bc_plist_create ( 1, RESEARCH );
       src := bc_src_create ( src_pl, bc_int, BC_ROLE_PIPE );
31     break;
     }
33   /* Start communication. */
     for ( month := JANUARY; month ≤ DECEMBER; month++ ) {
35     if ( bc_rank = ACCOUNTANT ∨ bc_rank = RESEARCH )
         bc_put ( sink, &salary[0], 1 ); /* Send amounts. */
37     else {
         bc_get ( src, &salary[0], 1 ); /* Receive amounts. */
39       printf ( "[%d] My salary: %d\n", bc_rank, salary[0] );
       }
41   }
     /* Destroy communication structure. */
43   if ( src_pl ) { bc_chan_destroy ( src ); bc_plist_destroy ( src_pl ); }
     if ( sink_pl ) { bc_chan_destroy ( sink ); bc_plist_destroy ( sink_pl ); }
45 }
```

Figure 3.12: β-channel implementation of Example 1.1.1. Each process first creates the β-channels before commencing communication. The β-channels created on each process are specific to the process, and represent the localised communication patterns expressed as the process' *roles*.

channels, or β-channels for brevity, which we define as follows:

Definition 3.8.1 (**Branching channel**)
A *branching channel*, or β-channel for brevity, is a data structure which encapsulates a dependency class, its projection, the associated role, and the communication specific parameters which affect the performance of communications upon activation (for example message buffers). ▌

Every process in the computation creates the necessary β-channel before commencing communications. A process only creates the β-channels that are defined in its corresponding partition. In the programming model, a source β-channel is created with the interface bc_src_create(); a sink β-channel is created with the interface bc_sink_create(). Both interfaces take the same parameters except for the message buffer size, which is passed while creating a sink β-channel.

We shall now implement Example 1.1.1. Figure 3.11 shows the dependency classes on each of the five partitions, T, A, S, R, and P. Each process creates one β-channel, each corresponding to the dependency class as shown. The complete implementation of Example 1.1.1 is shown in Figure 3.12. At lines 11–32, we create the β-channels. For every β-channel, a process list is first created to represent the projection of the dependency class which the β-channel encapsulates. This is done by invoking bc_plist_create(), as shown. The process list is then passed to either bc_src_create() or bc_sink_create() while creating the β-channel. We specify the role of the dependency class as another parameter. In order to specify the type of data that will be communicated through the β-channel, a data type is passed as the parameter. In addition, while creating a sink β-channel, the number of buffer units to be allocated for this β-channel is also passed. Once the β-channels are created, we commence the communication by activating the corresponding β-channels, as shown in lines 34–41. We invoke bc_put() on sink β-channels, and bc_get() on source β-channels. Finally, once the communications are over, the β-channels and their corresponding process lists are destroyed (lines 43–44).

Discussion

The β-channel abstraction model can be related directly to CSP and ACTOR systems, where the concept of β-channels extends the concept of a CSP channel, by allowing a process to communicate with multiple processes. In CSP, processes communicate with a one-to-one channel created by each of the process before commencing communication (see page 13). Through β-channels we can achieve the same effect by using β-channels with BC_ROLE_PIPE role. The advantage of β-channels, how-

ever, comes from the possibility of communicating with multiple processes while using a communication pattern—which is the fundamental basis for ACTOR systems. Another model similar to the actor system is the DAGGER approach [57], where executable components defined in the CHARM language are executed based on the availability of messages. In this model, each component behaves similar to ACTOR components, except for its similarity to data flow programming model [65].

Each of the DAGGER components in an application program does not initiate communications explicitly since they are scheduled for execution when the data required by that component has been received by the CHARM runtime system. This means that while some components are waiting for data, others can be executed, therefore avoiding the wastage of the processing elements. Although this provides a sense of asynchrony, as multiple components can execute simultaneously depending on data availability, debugging such programs will be complicated as this asynchrony is compounded by the inherent non-determinism of the system.

3.9 Practical advantages of the β-channel approach

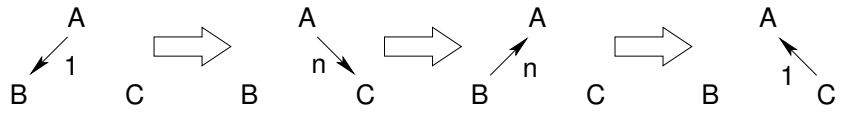
Most message passing systems provide handle based communications where all the communications are performed on a communication context defined by a communication handle. These contexts are usually implemented as opaque data structures upon which certain message passing actions are invoked to achieve data communications. What differentiates certain message passing systems from others is the manner in which a communication context is defined. For example, in LINDA [45], a communication context is defined by a *tuple space*, so that every communication is performed through a transparent implementation medium that provides certain access to tuples currently existing in the system. A process does not know who else is accessing the tuples. On the other hand, if we consider MPI systems [93], a communication context is defined by a *communicator* that provides a logical subset of the processes available to the programmer where all the processes in the group are assigned consecutive process ranks. It can therefore be argued that MPI systems provide more flexibility than LINDA because multiple communication contexts can be defined over the same set of processes. This comparison is important because the policy for defining communication contexts decides the practical aspects of the message passing system; and consequently affects the programming interfaces.

In the β-channel approach, the policy for defining a communication context is again different: instead of defining a context based on a global transparent handle, or through logical process group based handles, every communication context is de-

fined based on how the communication pattern is interpreted by the process under consideration (see Section 3.1). This means that every process knows about all the other processes with which it communicates, but this approach is more flexible than the MPI system because a process is not affected by, and does not affect, peer processes with which it does not perform any communication. This is how we resolve the *redundant acknowledgement problem* (see page 5).

Now, the question is how is the β-channel approach practically advantageous? To answer this question, let us study the following example.

Example 3.9.1
Assume three communicating processes, A, B and C, where A sends a data unit to B, and also sends n data units to C. From the data unit received, B generates n data units which are then sent to A. C, on the other hand, reduces the n data units received from A and sends the resulting data unit back to A. The following figure shows the communications,



For process group based systems, there exists only one communication context defined by the process group $\{A, B, C\}$. A communicator corresponding to this group provides the communication handle upon which communications are performed. The point to note here is that the communications performed during each stage exist under this single communicator, and therefore cannot be passively identified and referenced individually, without performing a communication; say for example, setting the size of the message buffer to be used for a particular communication.

Upon observation, we can see that in order to reduce the latency, process A can utilise message buffers. The number of buffers units allocated for each stage of the communications is not necessarily the same: sending one data unit to process B requires only one buffer unit; sending n data units to process C, on the other hand, requires n buffer units. In the above case with process group context handles, we cannot specify such specialised properties for each stage because they cannot be identified and referenced individually. This is where the β-channel approach gives more flexibility to the programmer, by allowing such specification of specialised communication structure specific properties.

In the β-channel approach, for each stage of communications, a communication context is defined by the β-channel created for performing the communication. In

Process	1	2	3	4
A	$1 \rightarrow B$	$n \rightarrow C$	$n \leftarrow B$	$1 \leftarrow C$
B	$1 \leftarrow A$	—	$n \rightarrow A$	—
C	—	$n \leftarrow A$	—	$1 \rightarrow A$

Table 3.1: The process specific communication contexts which correspond to the β-channels created for each stage of the communication. The notation $1 \rightarrow B$ means 1 data unit is sent to process B; $n \leftarrow C$ means that n data units are received from process C. '—' represents empty context where no context is defined by the process in that stage.

Table 3.1, we show the different contexts defined on each process. The notation $1 \rightarrow B$ means 1 data unit is sent to process B; $n \leftarrow C$ means that n data units are received from process C. '—' represents empty context, or no context is defined by the process in that stage. If we consider process A, for example, it defines four contexts: two, which correspond to the sink β-channels for sending data to B and C, and another two, which correspond to the source β-channels for receiving data from B and C. We can perform similar analyses for processes B and C.

Based on the analysis of Table 3.1, we can see that specifying specialised properties, such as message buffers, is very straightforward with the β-channel approach. All we have to do is pass the necessary buffer values while creating the β-channels. In Example 3.9.1, we can therefore specify the message buffer sizes on process A as:

```
bc_chan_t *bsink, *csink;
bc_plist_t *b, *c;
b := bc_plist_create (1, B); c := bc_plist_create (1, C);
bsink := bc_sink_create (b, bc_int, 1, BC_ROLE_PIPE);
csink := bc_sink_create (c, bc_int, n, BC_ROLE_PIPE);
bc_put (bsink, 1, &bdata); bc_put (csink, n, &cdata);
```

We can see from the above example that the β-channel approach provides a more flexible environment for the programmer. By allowing the programmer to have control over the communication properties, it allows for certain optimisations and further simplification, which we will discuss shortly. Before we discuss the interface optimisations for send-and-forget type communications let us first discuss the following β-channel properties :

- The *grouping property* of β-channels allows communications to be grouped with the same β-channel. Based on the role and the degree of the dependency class, all the communications represented by the dependency points are grouped as a single abstract entity.

- The *selectivity property* of β-channels allows selection of communications from a possible set of communications, represented by a dependency set. In combination with the grouping property, this property allows selection of dependency points from the dependency set so that they can be represented as a single activation unit.
- The *referencing property* of β-channels allows a group of selected communications to be referenced as a communication data structure. The most basic application of this property is the activation of a β-channel where communications are performed by activating a particular β-channel from a set of available β-channels.
- β-channels are *data typed*. This type is specified while creating the β-channel, and represents the type of data that can be communicated through the β-channel. Only data of types equivalent to the data type can be put into, or retrieved from that β-channel.
- The *existential property* of β-channels states that for a β-channel application program to be valid, all the communication structures should be created before activation. This means that all the β-channels that are required for a communication are available before commencing communications.

3.9.1 Avoiding intermediate memory copy

By combining the properties discussed in the previous section, we can achieve interface optimisations for send-and-forget type communications where a sending process does not reuse sent data. With the grouping property and the selectivity property, we are able to group a selection of dependency points. This, after translation to a β-channel, is available to the program as a passive data structure. Due to the referencing property, a β-channel can be acted upon during activation, or assigned certain properties (as discussed above in the case of message buffering). In addition, the β-channels can be manipulated directly, provided proper interfaces are available.

Programmer defined local variables abstract memory units to a particular data type. Hence, all memory units are potential local variables provided we can abstract them to a certain data type. When β-channels are created, message buffers are internally implemented by allocating memory units which are abstracted into buffer units of the β-channel data type. Therefore, buffer units within a β-channel are also potential local variables. By using the referencing property, we can therefore access these buffer units through proper interfaces.

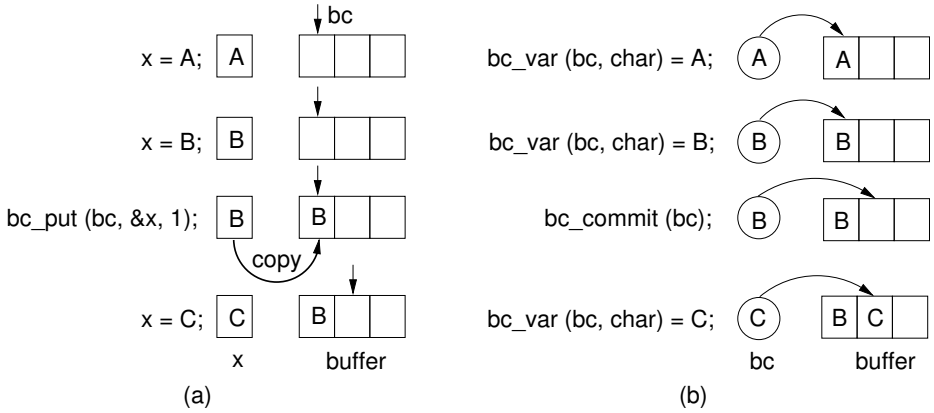


Figure 3.13: Message buffering, (a) with, (b) without, intermediate memory copy. buffer represents the internal message buffer units encapsulated by the β-channel bc.

The β-channel programming model provides three interfaces for accessing the buffer units directly.[4] They are: (1) bc_var(), (2) bc_vptr(), and (3) bc_commit(). The first two interfaces are macros which always point to the next available buffer unit. The last interface is an activation interface used to commit the current value of the buffer unit pointed to by the previous macros.

In Figure 3.13, we show message buffering with and without intermediate memory copy. The variable, x, in Figure 3.13.a is the local variable from which data is copied to a buffer unit during bc_put(). In Figure 3.13.b, instead of utilising a user defined local variable, through bc_var() we utilise a variable abstraction of a buffer unit. When the corresponding bc_commit() is invoked, the pointer to the buffer unit given by bc_var() is updated to the next valid buffer unit, also committing the previous value to the buffer. Through this mechanism, we have achieved message buffering without intermediate memory copy. In contrast to split-phase non-blocking APIs, more than one buffer unit can be utilised in the message buffer, while also simplifying programming which would be otherwise complicated by the multiple initiation-completion pairs. This is illustrated by the following example,

```
if ( bc_rank = PRODUCER ) {
    for ( i := 0; i < 10; i++ ) {
        bc_var ( a, int ) := compute ( data[i] );
        bc_commit ( a );
    }
} else for ( i := 0; i < 10; i++ ) bc_get ( a, &data[i], 1 );
```

In the above example, all the n buffer units are automatically utilised with a sin-

[4]The usage of the interfaces bc_var(), (b) bc_vptr(), and bc_commit() are thoroughly described in Section 4.2, and their implementation discussed in Section 5.5.1.

gle bc_var() and bc_commit() interface pair. The programming would have been complicated if MPI_Isend() and MPI_Wait() interfaces were used instead. In Section 4.3.4, we use these interfaces for implementing the Mandelbrot set task farm, where the input complex points to be communicated are set directly within the β-channel buffer units; the performance improvements are discussed in Section 6.2.1.

3.10 Summary

In this chapter, we have developed the β-channel abstraction model. We introduced new concepts for pattern abstraction based on our thesis that holistic patterns are best represented as implied runtime compositions of localised communication patterns (see Section 1.2).

The development of the β-channel abstraction model began with the re-analysis of what is meant by a communication pattern (see Section 3.1). To incorporate the findings of this analysis, we started with the existing fundamental concept of a control flow graph (see Section 3.2). We then introduced the concept of a *dependency point* (see Section 3.3) which defines the nodes of the control flow graph where data is sent or received from remote processes. As the aim is to integrate communication patterns within the message passing interfaces, we introduced the concept of a *dependency class* (see Section 3.4) which defines a logical grouping of dependency points lying on the same path within the control flow graph. These dependency classes formed the conceptual basis of the communication data structures, defined as *branching channels* (see Section 3.8), to which communication patterns, defined as *roles* (see Section 3.6) are assigned. To represent the holistic communication pattern manifested by the algorithm being implemented, we introduced the concept of a *communication structure* (see Section 3.7).

In relation to these new concepts, we have discussed how the *ambiguity* (see page 4), *loss of structural information* (see page 5), and *redundant acknowledgement* (see page 5) problems are resolved (see page 33 and page 52); the resolution of the *choice dilemma* (see page 4) and *performance portability* (see page 4) problems follow immediately because the ambiguity has been removed. We have also discussed the β-channel properties which allow communications to be selected, grouped, identified and referenced (see page 53). In order to demonstrate the practical advantages of these properties, we have discussed two applications: firstly, providing the flexibility to specify specialised communication properties, such as message buffers; and secondly, interface optimisation for *send-and-forget* communications, which avoids intermediate memory copy during buffering (see Section 3.9).

CHAPTER 4

Programming with communication structures

THIS CHAPTER discusses the programming model. Our main objective in this chapter is to understand the practical meanings of the abstraction concepts—in contrast to the conceptual developments presented in the previous chapter. We discuss here the practical considerations a programmer is faced with when applications using communication structures are being developed.

The exposition is divided into two major parts. The first part provides details of the application development process: starting from the parallelisation of an algorithm to the production of an executable parallel program. We introduce the two-phase application development process, and describe the related application programming interfaces with example usage notes. The rest of the first part is focused on justifying the qualitative advantages of the new approach by discussing implementations of several non-trivial message passing algorithms which manifest widely varying communication patterns. In the second part of this chapter, we discuss the relationship between the new model and skeletal parallel programming. We emphasise how the new model is advantageous for the implementation and deployment of algorithmic skeletons.

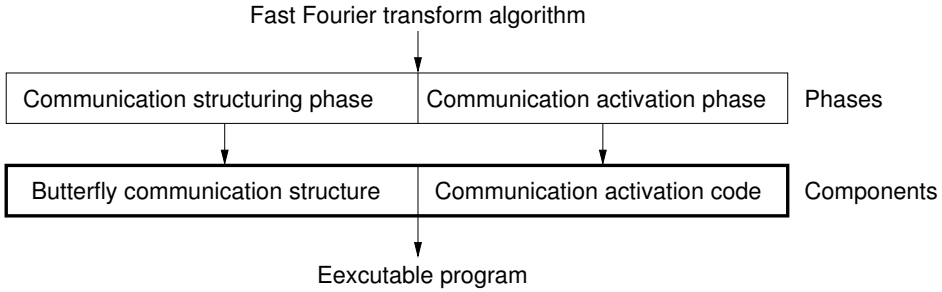


Figure 4.1: Two-phase application development. During the communication structuring phase, we derive the communication structure from the algorithm, i.e. *butterfly communication structure* for a fast Fourier transform algorithm. This is then translated to β-channels, and combined with the activation code generated during the communication activation phase. The highlighted box represents the application program.

4.1 Two-phase application development

After an algorithm is designed, it is implemented into an executable application by using a programming model. The programming model provides the programmer with the necessary tools for expressing the abstract representation of the algorithm into an executable form, which can then be executed on the runtime environment provided by the programming model.

Depending on the size and complexity of the application, application development is usually divided into several phases (for example, modular programming, object-oriented programming, etc.). Division into several phases allows programming concerns to be separated and handled independently of the others, thus reducing programming complexity. In the β-channel programming model, programming is divided into two main phases:[1] (1) communication structuring phase, and (2) communication activation phase (see Figure 4.1).

Communication structuring phase

During the *communication structuring phase*, the emphasis is on the understanding and analysis of the communication patterns manifested by the algorithm. From the analysis, communication structures which represent these patterns are derived. In addition to representing inter-process data dependencies, the communication structures also represent the β-channels that should be created before commencing data

[1] Although, each of these phases may be divided further into modules etc., we will not consider such divisions here. It is up to the programmer to choose the best approach for further division of these phases, if necessary.

communications during program execution. The end result of this phase is a collection of β-channels defined by each of the participating processes, which collectively represents the communication patterns manifested by the algorithm. Any communication performed over this set of β-channels therefore represents the pattern, automatically—without the need for further programmer intervention.

In Figure 4.1, for example, we implement the fast Fourier transform algorithm (discussed more thoroughly in Section 4.3.2). During the communication structuring phase, the communication structures that define the butterfly communication pattern is derived and translated to the necessary β-channels. The final result of this phase is a 'passive' network of β-channels through which data can be communicated in a butterfly communication pattern. This, however, does not constitute the actual communications required during the computations. They should therefore be activated in order to perform actual communications, which is done during the communication activation phase.

Communication activation phase

During the *communication activation phase*, parts of the program that are necessary for actually communicating data are programmed. The β-channels resulting from the communication structuring phase are *passive*: they only define how a communication can proceed during execution; and therefore do not mean anything during the execution unless they are activated. Hence, in order to communicate data, the β-channels are activated through a set of interfaces which provides the actions.

The set of 'actions' that can be applied to a β-channel are: putting data into the β-channel, and getting data from the β-channel. These two actions correspond to whether the process is acting as the producer, or the consumer: 'put' actions for producer processes, and 'get' actions for consumer processes.

In Figure 4.1, as discussed in the previous section, the butterfly communication structure for the fast Fourier transform is translated into a network of β-channels. During the communication activation phase, we utilise these β-channels by applying put or get actions. Within the computation, the corresponding sink β-channels are activated with `bc_put()` where data are sent, and the source β-channels are activated with `bc_get()` where data should be received before further computations.

In essence, the above two divisions follow the fundamental concept of structured programming, where application development begins by defining data structures (such as queues, lists, stacks etc.) and then applying actions (such as 'push' and 'pop') to these data structures based on what is required by the algorithm; thus resulting in an executable program [102].

4.2 Application programming interfaces

This section describes the application programming interfaces for developing message passing parallel programs with communication structures. We organise this section so that related interfaces are described together. Wherever possible, we give running example notes to briefly demonstrate their usage. Further discussion of their usage when applied to the implementation of real algorithms will be discussed in Section 4.3.

In order to avoid namespace corruption, the interfaces and the data structures are prefixed with 'bc_', and constants with 'BC_'. It is also worth noting that, while naming functions and variables we follow a convention similar to the POSIX standard [64]. Unless otherwise stated, all of the programming interfaces return an integer error code.

Initialisation and finalisation

A β-channel application program should initialise the programming library first. This allocates the data structures necessary for managing the runtime system. In order to deallocate these resources at the end of the computations and communications, the application program should finalise the library before returning. The application programming interfaces for initialisation and finalisation are,

int **bc_init** (int flag);

Initialises the programming library. bc_init() should be invoked before using any of the library functions and associated data structures; and this should be done only once. The flag gives the options for library functionalities. After initialisation, two values are defined on every process: bc_size and bc_rank, where bc_size gives the number of processes available during initialisation, and bc_rank gives the rank of the process in the process ensemble. All the available processes are ranked consecutively starting at zero; and every process retains its rank until bc_final().

int **bc_final** (void);

Finalises the programming library. This should be invoked at the end of the application to deallocate internal data structures created during bc_init().

Process list management

The *process list* is a data structure defined by a process to identify the remote processes with which it communicates through a β-channel. They represent the pro-

cesses with which the process should interact in order to realise its communication role. Process lists exist independently of any β-channel, therefore, they can be shared by multiple β-channels. The application programming interfaces for managing process lists are,

bc_plist_t *__bc_plist_create__ (int num, ...);

Returns a valid process list with |num| processes. The number of processes listed by the variable arguments should equal |num|. If num > 0, the process list is created with the listed processes. If num < 0, the process list is created with all processes available after bc_init(), *excluding* those that are listed. If num $= 0$, NULL is returned, meaning error. For example, if the available processes are $\{0,1,2,3,4\}$, the following will create a process list a with processes $\{0,1,2\}$, and process list b with processes $\{1,3,4\}$.

```
bc_plist_t *a, *b;
a := bc_plist_create (3, 0, 1, 2);
b := bc_plist_create (-2, 0, 2);
```

bc_plist_t *__bc_plist_create_empty__ (int num);

Returns an empty process list with enough placeholders for num > 0 processes; If num $\leqslant 0$, NULL is returned, which means error. The returned process list is *not valid* and should not be used before setting the process ranks with bc_plist_set(). bc_plist_create_empty() is used when processes need to be assigned dynamically. For example, an empty process list a for holding 10 processes is created as,

```
bc_plist_t *a;
a := bc_plist_create_empty (10);
```

int __bc_plist_set__ (bc_plist_t *plist, int loc, int proc);

Sets or resets a process in a process list. Processes are ordered within a process list with consecutive indices starting at 0. bc_plist_set() sets the process at the placeholder indexed by loc in the plist with the process rank proc. The process at loc is always overwritten.

A process list should only be set or reset when no β-channel is using it. If bc_plist_set() is invoked without satisfying this condition, the function returns immediately without having any effect. For example, a process list a of 10 consecutive even processes starting with process 2 is created as,

```
bc_plist_t *a;
a := bc_plist_create_empty (10);
for (i := 0, j := 2; i < 10; i++, j += 2) bc_plist_set (a, i, j);
```

Provided no β-channel is using the process list a created above, the last 5 even processes can be reset to 5 consecutive odd processes starting at 3 as follows,

```
for (i := 5, j := 3; i < 10; i++, j += 2) bc_plist_set (a, i, j);
```

int **bc_plist_split** (bc_plist_t *plist, int num, bc_plist_t *new[]);

Splits a process list plist into num process lists. The newly created process lists will be stored in new; which should be provided by the programmer. Each new process list is assigned processes from the ordered set in plist. If an equal division of processes cannot be made, $\lfloor size/num \rfloor$ processes will be assigned to each of the first $num-1$ process lists, and the remaining processes will be assigned to the last process list. Here $size$ gives the number of processes in plist. All of the new process lists are disjointed so that no two process lists will have the same process. Also, plist is left unchanged. For example, the process list $a = \{0,1,2,3,4,5,6\}$ is split into 3 new process lists, $s[0] = \{0,1\}$, $s[1] = \{2,3\}$, and $s[2] = \{4,5,6\}$ as follows,

```
bc_plist_t *a, *s[3];
a := bc_plist_create (7, 0, 1, 2, 3, 4, 5, 6);
bc_plist_split (a, 3, s);
```

bc_plist_t ***bc_plist_union** (int num, bc_plist_t *plists[]);

Returns the *set union* of the num process lists pointed to by plists. When the new process list is created, the supplied process lists are left unchanged. For example, the set union $u = \{0,1,2,3,4,5\}$ of the two process lists $s[0] = \{0,1,2,5\}$ and $s[1] = \{0,1,3,4,5\}$ is obtained as follows,

```
bc_plist_t *u, *s[2];
s[0] := bc_plist_create (4, 0, 1, 2, 5);
s[1] := bc_plist_create (5, 0, 1, 3, 4, 5);
u := bc_plist_union (2, s);
```

bc_plist_t ***bc_plist_isect** (int num, bc_plist_t *plists[]);

Returns the *set intersection* of the num process lists pointed to by plists. When the new process list is created, the supplied process lists are left unchanged. For example, the set intersection $i = \{0,1,5\}$ of the two process lists $s[0] = \{0,1,2,5\}$ and $s[1] = \{0,1,3,4,5\}$ is obtained as follows,

```
bc_plist_t *i, *s[2];
s[0] := bc_plist_create (4, 0, 1, 2, 5);
s[1] := bc_plist_create (5, 0, 1, 3, 4, 5);
i := bc_plist_isect (2, s);
```

Flag	bc_plist_t *	Description
BC_PLIST_SELF	bc_plist_self	Self reference.
BC_PLIST_ALL	bc_plist_all	All processes including self.
BC_PLIST_XALL	bc_plist_xall	All processes excluding self.
BC_PLIST_ODD	bc_plist_odd	All odd processes excluding self.
BC_PLIST_EVEN	bc_plist_even	All even processes excluding self.
BC_PLIST_PRED	bc_plist_pred	All preceding processes.
BC_PLIST_SUCC	bc_plist_succ	All succeeding processes.

Table 4.1: Builtin process lists that can be used immediately after initialisation.

bc_plist_t *__bc_plist_diff__ (bc_plist_t *a, bc_plist_t *b);

Returns the *set difference* of the process lists a and b. When the new process list is created, the supplied process lists are left unchanged. For example, the set difference $d = \{2,6,7,9\}$ of the process lists $a = \{0,1,2,5,6,7,9\}$ and $b = \{0,1,3,4,5\}$ is obtained as follows,

```
bc_plist_t *d, *a, *b;
a := bc_plist_create (7, 0, 1, 2, 5, 6, 7, 9);
b := bc_plist_create (5, 0, 1, 3, 4, 5);
d := bc_plist_diff (a, b);
```

int __bc_plist_destroy__ (bc_plist_t *plist);

Destroys the process list plist. The actual deallocation of resources is handled by the runtime system when it is safe to proceed.

When process lists are created or split, the order of the processes in the initial process lists is always preserved in the resulting process lists.

Builtin process lists

This section describes some of the builtin process lists (see Table 4.1). These process lists are optional functionalities provided by the runtime system, and can be activated by setting the appropriate flag value before bc_init().

bc_plist_self contains the rank of the process, and is used for self referencing. bc_plist_all refers to the rank of all the bc_size processes available during bc_init(). bc_plist_xall refers to the ranks of all the other (bc_size − 1) processes available during bc_init(), excluding the rank of the invoking process. bc_plist_odd and bc_plist_even respectively refer to all of the odd and even ranked processes available during bc_init(), excluding the rank of the invoking process. bc_plist_pred refers to the ranks of all the processes in the process ensemble whose process ranks are less than the rank of the

bc_dtype_t *	C data type
bc_char	char
bc_uchar	unsigned char
bc_short	short
bc_ushort	unsigned short
bc_int	int
bc_uint	unsigned int
bc_float	float
bc_long	long
bc_ulong	unsigned long
bc_double	double
bc_long_double	long double

Table 4.2: Builtin β-channel data types, and corresponding C language data type.

invoking process. Similarly, bc_plist_succ refers to the ranks of all the processes in the process ensemble whose process ranks are greater than the rank of the invoking process.

When any of these builtin process lists is required, it can be requested by setting the appropriate value of flag during initialisation. If more than one process list is desired, flag is set to the bitwise 'OR'ing of the appropriate flags. For example, to broadcast data to all of the succeeding processes, and to sum reduce data from all of the preceding processes, one can use bc_plist_pred and bc_plist_succ by first requesting them during initialisation with bc_init(BC_PLIST_PRED | BC_PLIST_SUCC).

Builtin process lists need not be destroyed by the programmer because they are managed by the runtime system, and hence are destroyed automatically during finalisation. Invoking bc_plist_destroy() on builtin process lists does not have any effect, and therefore the function returns immediately.

β-channel data types

This section discusses the β-channel data types which specialise a communication structure by providing information on the type of data that can be communicated through the communication structure. Every β-channel is associated with a data type, and this specifies the type of data which can be sent or received through the β-channel. Because communication structure only defines the manner of interaction between processes, it is important to specialise this with information about the actual data that will be communicated. For example, if we desire a pipeline communication structure through which integer data are communicated, we must specialise

this structure by specifying integer data types for the corresponding β-channels. It should be noted here that we only discuss the case where a β-channel is associated with one data type only; however, this can be extended to support mixed data types where a β-channel can be used for transferring different types of data depending on, for example, the runtime execution instance.

Builtin data types

The programming library defines the builtin data types given in (see Table 4.2). They can be used immediately following a successful bc_init().

Custom data types

Some applications may require transfer of data that cannot be represented with the builtin data types; or, it may be desirable to communicate data of different data types packed as a single data unit. For such applications, a custom data type should be created. The application programming interfaces for managing custom data types are,

bc_dtype_t *__bc_dtype_create__ (size_t size);

Creates a custom data type that can be represented in the virtual memory with size bytes. For example, a custom data type n for communicating a C programming language *structure* custom with two members: number of type int and a char array name of length 10 is created as,

```
struct custom { int number; char name[10]; } ;
bc_dtype_t *n;
n := bc_dtype_create (sizeof (struct custom));
```

Newly created custom data types can be shared by multiple β-channels.

int __bc_dtype_destroy__ (bc_dtype_t *dtype);

Destroys a custom data type. Custom data types are not automatically destroyed by the runtime system. Hence, by invoking bc_dtype_destroy(), a process should explicitly request the runtime system for deallocation. Once a request for deallocation is received, the runtime system performs the deallocation when it is safe to proceed.

β-channel roles

This section describes the β-channel roles currently provided by the programming library. These roles define the manner in which producer and consumer processes

interact during a communication. Figure 4.2 shows the semantics of the β-channel roles in terms of graphical representations.

BC_ROLE_PIPE

When a producer sink β-channel with BC_ROLE_PIPE role is activated with bc_put(), data from the application buffer are copied into the internal buffer that is associated with the sink β-channel. When the corresponding source β-channel on the consumer is activated, using bc_get(), data from this internal buffer is transferred to the application buffer of the consumer.

When a source β-channel with BC_ROLE_PIPE role is activated with bc_get(), a data transfer request is sent to the producer associated with that β-channel and the consumer waits until the data have been received successfully.

BC_ROLE_REPLICATE

When a sink β-channel with BC_ROLE_REPLICATE role is activated with bc_put(), data from the application buffer are copied into the shared internal buffers associated with the β-channel. When the corresponding source β-channels on the consumers are activated, data from this internal buffer is transferred to the application buffer of the consumers. Even though the effect of this role is that of a data broadcast, the name BC_ROLE_REPLICATE is chosen because the producer does not explicitly broadcast data to all of the consumers. Instead, data in the internal buffer are shared by the consumers during retrieval, which results in the replication of the same data on all the consumers.

BC_ROLE_SPREAD

When a sink β-channel with BC_ROLE_SPREAD role is activated with bc_put(), data which are unique to each consumer are copied from the application buffer into the respective buffers associated with each consumer of that β-channel. Upon activation of the corresponding source β-channels on the consumers, data from the corresponding internal buffer are transferred to the application buffer of the requesting consumer. The manner in which data are copied from the application buffer depends on the ordering of processes within the process list associated with the sink β-channel. This decides which consumer gets what data. Therefore, the first data goes to the first process in the process list, and so on. The name BC_ROLE_SPREAD is chosen instead of BC_ROLE_SCATTER in order to reflect this ordering.

BC_ROLE_FARM

When a sink β-channel with BC_ROLE_FARM role is activated with bc_put(), data from the application buffer are copied into the shared internal buffer asso-

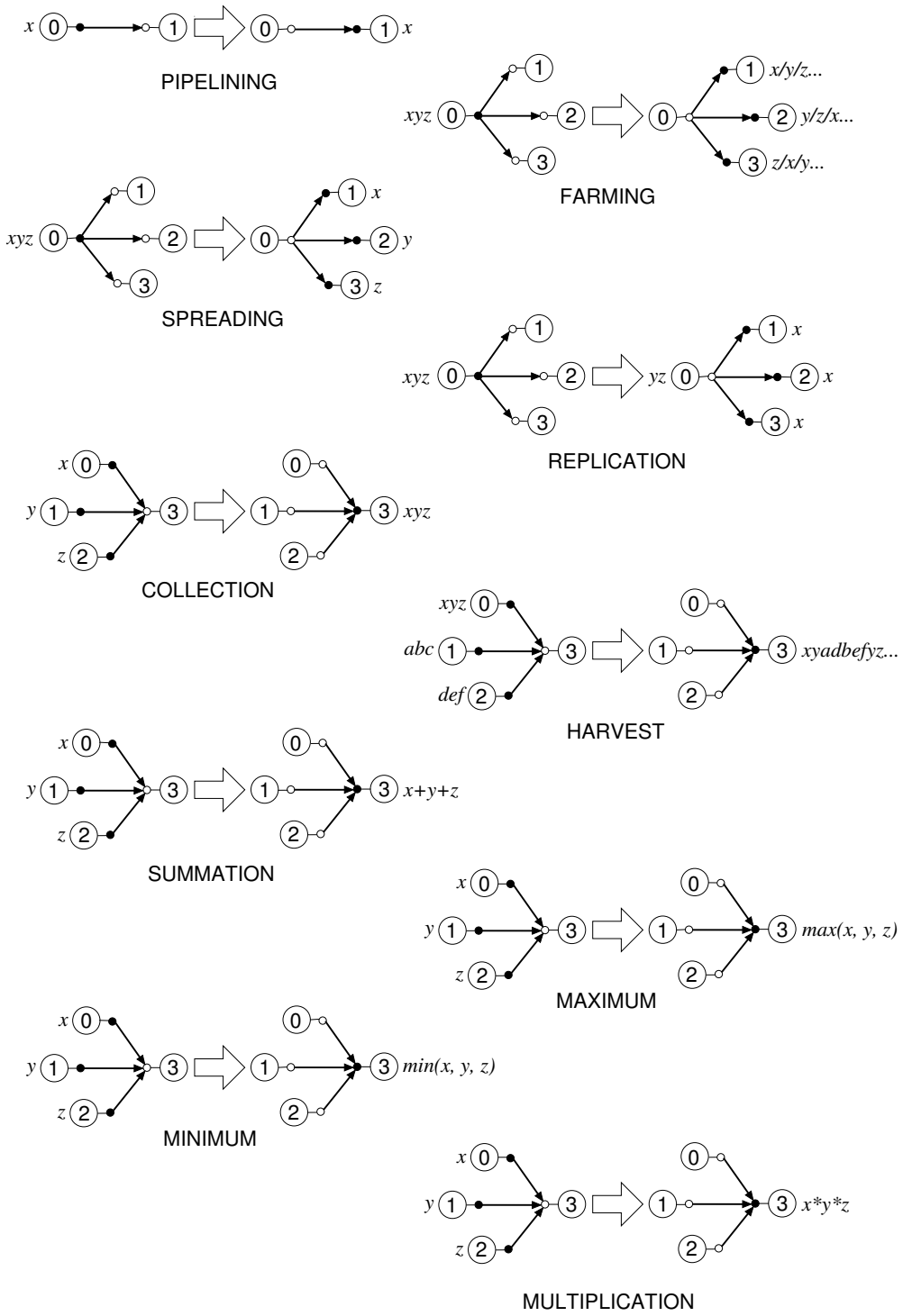


Figure 4.2: Semantics of the β-channel roles. • represents activation of a β-channel.

ciated with the β-channel. From the producer's perspective, the effect is similar to that of BC_ROLE_PIPE, except for the internal events which take place when the corresponding source β-channels are activated. Upon activation of the corresponding source β-channels on the consumers, data from the internal buffer gets transferred to the application buffer of the consumer. The transferred data, however, are considered unique because only the receiving consumer gets the data (exclusive). Which consumer gets what data is non-deterministic, and depends on who requested the data first.

BC_ROLE_COLLECT

When a consumer source β-channel with BC_ROLE_COLLECT role is activated with bc_get(), data is transferred from the internal buffer of all the corresponding producer β-channels to the consumer application buffer. Data received from each of the producers are treated uniquely, and are ordered in the application buffer according to the ordering of the producers in the process list associated with the source β-channel. If the source β-channel has n producers, activation of this β-channel will therefore result in the transfer of n data units from the producers into the application buffer, where received data are ordered based on the ordering of the n producers in the associated process list.

BC_ROLE_HARVEST

When a consumer source β-channel with BC_ROLE_HARVEST role is activated with bc_get(), data is transferred from the internal buffer of the sink β-channel to the consumer application buffer, by choosing the producer based on data availability. This is similar to the BC_ROLE_COLLECT role, except for the non-determinism of the producer from which the data is received. In practice, this role can be considered as the receiver counterpart to the BC_ROLE_FARM role, where decisions are made according to when the request for data arrived.

BC_ROLE_REDUCE_SUM

When a consumer source β-channel with BC_ROLE_REDUCE_SUM role is activated with bc_get(), the sum of the data transferred from the internal buffers of all the corresponding producer β-channels is stored in the application buffer of the consumer. If the source β-channel has n producers, activation of this β-channel will therefore result in the sum reduction of the n data units transferred from the producers; finally storing the sum into the application buffer. Only the sum is stored in the application buffer, and the intermediate data received from the producers are not available to the consumer.

BC_ROLE_REDUCE_MUL

Similar to BC_ROLE_REDUCE_SUM, but performs multiplicative reduction.

BC_ROLE_REDUCE_MAX

Similar to BC_ROLE_REDUCE_SUM, but calculates the maximum value.

BC_ROLE_REDUCE_MIN

Similar to BC_ROLE_REDUCE_SUM, but calculates the minimum value.

The β-channel roles described here only constitute the most common roles found in parallel algorithms. Additional roles can be introduced without affecting the existing interfaces and data structures. This makes the programming model highly extensible, and easily manageable. For example, the last four roles reduce data received from the sender processes. Such roles incorporate trivial computations within the communications. In situations where the programmer wishes to devise a custom operator function for the data reduction, one can introduce a new role, say BC_ROLE_REDUCE_OPT, for supporting such customisations. In the next section, we shall discuss how the functionality of the β-channel programming model can be extended with new roles.

Extending the β-channel roles

In this section, we discuss extension of the β-channel programming model by introducing a new role which allows a programmer to specify custom operator functions for data reduction. We shall refer to this role as BC_ROLE_REDUCE_OPT.

All computations that are incorporated within the communications are performed as part of the communication, transparent from the application programmer. In all of the last four roles in the previous section, the computations are performed after the data has been received from all the sender processes. In principle, therefore, the implementation of these roles can be separated into communications with the BC_ROLE_COLLECT role, following which computations on the data that has been received (e.g. summation, finding the maximum etc.) are performed.

The first part of the extension involves adding the name of the new role into the roles database so that source β-channels using this role can be created with bc_src_create(). The second part of the extension involves implementing the interface function that will provide a functionality similar to the one provided by the BC_ROLE_COLLECT role; so that all the data from the sender processes is received into a temporary buffer. The third, and final, part of the extension is invoking the custom function provided by the application programmer, over the set of data units that have been collected in the temporary buffer.

To support the BC_ROLE_REDUCE_OPT role, we need a way for the program-

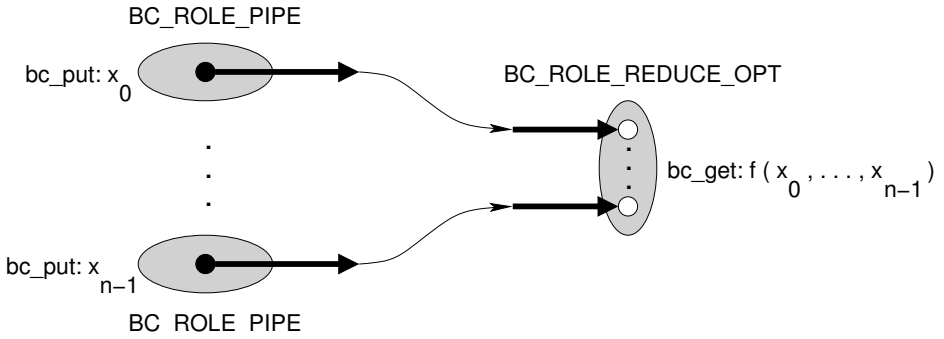


Figure 4.3: Extending the set of β-channel roles by introducing a new role which supports specification of custom operator functions for data reduction. A new role, BC_ROLE_REDUCE_OPT, is introduced. This role invokes the custom function, $f(x_0,\ldots,x_{n-1})$, during activation.

mer to express the function $f(x_0,\ldots,x_{n-1})$. As shown in Figure 4.3, this function can then be invoked when a source β-channel with BC_ROLE_REDUCE_OPT role is activated. To specify the custom operator function without changing the existing interfaces, we define a function pointer bc_operator with the prototype:

```
typedef void (*bc_opt_t) (void *buffer, int count);
```

where buffer is a pointer to a temporary buffer where received data is collected, and count gives the number of data units that have been collected in buffer.

As an example, to create a custom operator function which adds integer values received from all the even indexed processes in the process lists, while subtracting integer values received from the odd indexed processes, we may define a function, custom_operator, and assign it to bc_operator as follows:

```
void custom_operator (void *buffer, int count) {
    int i, value;
    for (i := 0; i < count; i++) {
        if (i % 2) value -= *((int *) buffer + i);
        else value += *((int *) buffer + i);
    }
    *(int *) buffer := value;
}
bc_operator := custom_operator;
```

Source β-channels using the BC_ROLE_REDUCE_OPT role can now be created using the bc_src_create() interface (see next section). When such a β-channel is activated, the runtime system will first receive all the values from the remote processes listed in the process lists corresponding to the β-channel, and then invoke the function assigned to bc_operator, which in this case is the function custom_operator: the

appropriate parameters—pointer to the temporary buffer and the number of data units available on that buffer—are supplied by the runtime system.

By following procedures similar to the ones discussed above, we can extend the β-channel programming model with more complex roles. As we can observe, the addition of new roles does not affect the existing interfaces and roles: it only increases the functionality because the newly added role can be used in conjunction with existing roles. It should be noted, however, that extensions with custom operator functions are not type safe. It is the programmers' responsibility to ensure that the temporary buffer supplied by the β-channel runtime system is dereferenced appropriately to the β-channel data type within the custom operator function.

β-channel management

This section describes β-channel management. After identifying the communication structures necessary for implementing a given algorithm, they are translated into β-channels by specifying the process list, role, data type and buffer characteristics. Following the producer-consumer relationship, producer processes create sink β-channels, while consumer processes create source β-channels.

bc_chan_t *__bc_src_create__ (bc_plist_t *prod, bc_dtype_t *dtype, bc_role_t role);

Returns a source β-channel. prod is the process list which gives the set of producers on which the consumer depends for data. dtype gives the type of data that can be received through this source, and the role defines the manner in which the data received from the producers are stored in the application buffer. For example, a source β-channel of integer data type which sum reduces data received from all the odd ranked processes, excluding the invoking process, is created as follows,

```
bc_chan_t *src;
src := bc_src_create (bc_plist_odd, bc_int, BC_ROLE_REDUCE_SUM);
```

bc_chan_t *__bc_sink_create__ (bc_plist_t *cons, bc_dtype_t *dtype, int bu, bc_role_t role);

Returns a sink β-channel. cons is the process list which gives the consumer processes depending on this β-channel for data. dtype is the type of data that can be sent through this β-channel, and bu gives the number of buffer units to be allocated for the internal buffer. The role defines the manner in which data from the application buffer are stored into the internal buffer associated with the β-channel. For example, a sink β-channel with 10 buffer units for sending integer data that are replicated on all the succeeding processes is created as

follows,

```
bc_chan_t *sink;
sink := bc_sink_create (bc_plist_succ, bc_int, 10, BC_ROLE_REPLICATE);
```

int **bc_chan_destroy** (bc_chan_t *bc);

Destroys a β-channel. Actual deallocation of resources is handled by the runtime system.

All the interfaces described hitherto are executed locally, and therefore are not influenced by any of the remote processes. Execution of these interfaces are therefore guaranteed to be asynchronous.

Planarity of dependency edges

In the prototype implementation of the runtime system, all the β-channels that are created on a process are assigned a unique identification tag internally. These tags are used to resolve the sink-to-source dependency edges during communications. Using this tag assignment policy allows asynchronous execution of the bc_put() and bc_get() interfaces. However, in order to perform the resolutions properly it is necessary that all β-channel creations observe the *planarity condition*, defined as follows:

Definition 4.2.1 (**Planarity condition**)
When β-channels are created, all the dependency edges between any *two* processes formed by a combination of source and sink β-channels should be arranged in such a manner that no two dependency edges, represented by a *straight* directed edge, with the same direction of data flow cross each other. If they do cross, this should be resolved by reordering the sequence in which the β-channels are created on either of the two processes. This condition, however, does not apply to dependency edges which have different directions of data flow, and does not affect the order in which the β-channels are used during the communication activation phase. ▌

In Figure 4.4, for example, we have two communicating processes A and B. In case (a), A creates two sink β-channels m and n, when B creates two source β-channels x and y sequentially. Both dependency edges (m, y) and (n, x) have the same direction of data flow, and they are non-planar. We must resolve this crossing by reordering the creation of β-channels in one of the processes. As shown in case (b), we resolve the crossing by reordering the creation of the source β-channels on B so that β-channel y is created before x. The crossing can also be removed

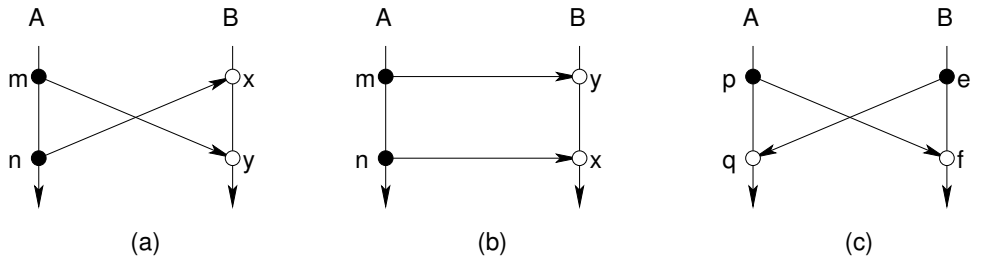


Figure 4.4: The internal tag assignment policy requires that the 'planarity condition' is satisfied. (a) Non-planar dependency edges where the two edges (m, y) and (n, x) cross each other, and (b) the crossing in (a) is resolved by reordering the β-channel creations on $\mathcal{P}_B$. (c) non-planar dependency edges with opposite direction of data flow can be left as they are.

by changing the order in which m and n are created on A, so that creation of n precedes that of m.

On the other hand, the two dependency edges (p, f) and (e, q) as shown in Figure 4.4.c can be left as they are because the direction of the data flow is opposite: for dependency edge (p, f), the directed edge points from process A to process B, while the directed edge for the dependency edge (e, q) points from process B to process A.

In Section 5.3.3, we shall discuss the reasons for the 'planarity condition'.

β-channel activation

This section describes how a communication structure is activated for communicating data. After a communication structure is translated into the corresponding β-channels, the producer activates the sink β-channel while the consumer activates the source β-channel.

When a sink β-channel is activated, data from the application buffer is transferred into the internal message buffer associated with that β-channel. We perform this action by invoking the bc_put() interface on the appropriate sink β-channel.

int **bc_put** (bc_chan_t *sink, void *abuff, int dunit);

Activates a sink β-channel. This transfers dunit data units from the application buffer, abuff, to the internal message buffer of sink. One important condition is that the β-channel buffer should have at least dunit buffer units. For example, in the following code segment, bc_put() transfers 4 data units from the application buffer value into the internal buffer of sink.

```
bc_chan_t *sink; bc_plist_t *cons;
int value[] := {0, 1, 2, 3};
cons := bc_plist_create (4, 1, 2, 3, 4);
```

```
sink := bc_sink_create (cons, bc_int, 4, BC_ROLE_REPLICATE);
bc_put (sink, value, 4);
bc_chan_destroy (sink); bc_plist_destroy (cons);
```

When a source β-channel is activated, a data transfer request is sent to the producers. The data are then received into the application buffer. We perform this action by invoking the bc_get() interface on the appropriate source β-channel.

int **bc_get** (bc_chan_t *src, void *abuff, int dunit);

Activates a source β-channel. This retrieves dunit data units from the source β-channel src, and stores the received data in the application buffer, abuff. For example, in the following code segment, bc_get() stores 4 data units in the application buffer value, where the stored data are the sum reductions of the data received from the producers for each data unit requested.

```
bc_chan_t *src; bc_plist_t *prod;
int value[4];
prod := bc_plist_create (4, 1, 2, 3, 4);
src := bc_src_create (prod, bc_int, BC_ROLE_REDUCE_SUM);
bc_get (src, value, 4);
bc_chan_destroy (src); bc_plist_destroy (prod);
```

Interfaces for avoiding intermediate memory copy

The interfaces for avoiding intermediate memory copy for send-and-forget type communications are as follows:

bc_var (bc_chan_t *sink, c_type type);

bc_var() is a macro which expands to a valid buffer unit within the sink β-channel buffer. The valid buffer unit is dereferenced to the C programming language data type, type: equivalent to a programmer defined variable. For example, the following dereferences the β-channel buffer unit as an integer variable.

```
bc_chan_t *sink;
sink := bc_sink_create (bc_plist_xall, bc_int, 4, BC_ROLE_REPLICATE);
bc_var(sink, int) := 10;
bc_chan_destroy (sink);
```

The data type type which is passed to bc_var() ensures that the buffer unit is properly dereferenced. This is necessary because the buffer units are not associated with any data type on its own; they are resolved when the β-channel is activated. In this case, because we are accessing the internals of the sink

β-channel without actually activating it, we have to make sure that the buffer unit is properly dereferenced.

bc_vptr (bc_chan_t *sink, c_type type);

bc_vptr() is similar to bc_var(). However, instead of abstracting a valid buffer unit to a variable, it expands to a type variable pointer. This is required for sending an array of data. For example, in the following code segment we create a sink which can accommodate 4 buffer units, with each buffer unit storing an array of 10 integers. Once the sink has been created, we fill the values for one buffer unit with 10 integers using bc_vptr().

```
int i; bc_dtype_t *dtype; bc_chan_t *sink;
dtype := bc_dtype_create (10, sizeof(int));
sink := bc_sink_create (bc_plist_xall, dtype, 4, BC_ROLE_REPLICATE);
for (i := 0; i < 10; i++) *(bc_vptr(sink, int) + i) := i;
bc_chan_destroy (sink);bc_dtype_destroy (dtype);
```

int **bc_commit** (bc_chan_t *sink);

bc_commit() commits the value in the variable abstraction to a buffer unit value by updating the sink buffer pointer to the next valid buffer unit. Upon return, bc_var() and bc_vptr() points to a new buffer unit. Until we issue bc_commit(), the values in the current buffer unit—which is pointed to by bc_var() and bc_vptr()—cannot be sent to any of the remote receivers. For example, in the following code, we first fill the 10 integers for the first data unit, and then commit the value so that we can fill up the next data unit.

```
int i, j; bc_dtype_t *dtype; bc_chan_t *sink;
dtype := bc_dtype_create (10, sizeof(int));
sink := bc_sink_create (bc_plist_xall, dtype, 4, BC_ROLE_REPLICATE);
for (i := 0; i < 4; i++) {
    for (j := 0; j < 10; j++) *(bc_vptr(sink, int) + j) := i*j;
    bc_commit(sink);
}
bc_chan_destroy (sink); bc_dtype_destroy (dtype);
```

We illustrate the above three interfaces during the implementation of the pipeline skeleton interface (see Section 4.4.2), and also during the implementation of the non-deterministic Mandelbrot set task farm (see Section 4.3.4). Implementation details are discussed in Section 5.5.1, and latency improvement in Section 6.2.1.

4.3 Implementing common algorithms

This section demonstrates the β-channel programming model by applying the concepts and programming interfaces for implementing five common non-trivial parallel algorithms. These algorithms are chosen because of the communication patterns they manifest, so that subtle features of the programming model can be described. For each of the algorithms, we begin the discussion with a brief definition of the algorithm and related terms, followed by a description of the parallelisation approach. We then discuss the β-channel implementation.

4.3.1 Gaussian elimination

In scientific problems, it is often necessary to solve a system of linear equations. For realistic problems, such systems of linear equations are often quite large; therefore solving these systems is computationally demanding. In order to provide the computational power required to solve such systems, parallel computing systems are employed. Many parallel algorithms have been designed to solve a system of linear equations, and Gaussian elimination is a well-known algorithm.

Definition 4.3.1 (**Linear equation**)
A *linear equation* on n variables $x_1, \ldots, x_n$ is an equation which can be expressed as $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$, where $a_1, \ldots, a_n$ are the coefficients of the equation, and b is a constant. ▌

Definition 4.3.2 (**System of linear equations**)
A *system of linear equations* is a finite set of linear equations on n variables $x_1, \ldots, x_n$ which can be solved to give a set of constants $s_1, \ldots, s_n$, also known as the *solution set*, which, when substituted for the variables, $x_i = s_i$ for $1 \leqslant i \leqslant n$, satisfies all equations in the system of linear equations. ▌

A system of linear equations on n variables can also be represented as $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is an $n \times n$ matrix containing the coefficients a_{ij} of all the equations in the system, and $\mathbf{x}$ and $\mathbf{b}$ are $n \times 1$ vectors respectively storing the values of x_i and b_i, for $1 \leqslant i, j \leqslant n$. The locations and values of non-zero elements in $\mathbf{A}$ determine the complexity of solving these equations: for a sequential algorithm, the time complexity is generally $O(n^3)$; however, upper triangular and lower triangular systems can be solved in $O(n^2)$ with a sequential algorithm [86].

Definition 4.3.3 (**Upper and Lower triangular**)
An $n \times n$ matrix is *upper triangular* if for all $1 \leqslant i, j \leqslant n$, $a_{ij} = 0$ when $i > j$. An $n \times n$ matrix is *lower triangular* if for all $1 \leqslant i, j \leqslant n$, $a_{ij} = 0$ when $i < j$. ▌

```
1  void gaussian_sequential ( int A[][], int b[], int c[] , int n) {
       int i, j, k;
3      for ( i := 0; i < n; i++ ) {
           for ( j := i + 1; j < n; j++ )
5              A[i][j] := A[i][j] / A[i][i];
           c[i] := b[i] / A[i][i];
7          A[i][i] := 1;
           for ( k := i + 1; k < n; k++ ) {
9              for ( j := i + 1; j < n; j++ )
                   A[k][j] := A[k][j] − A[k][i] ∗ A[i][j];
11             b[k] := b[k] − A[k][i] ∗ c[i];
               A[k][i] := 0;
13         }
       }
15 }
```

Figure 4.5: A sequential implementation of the Gaussian elimination algorithm for reducing a system of linear equations to an upper triangular form. This can be parallelised in the outermost loop (line 3).

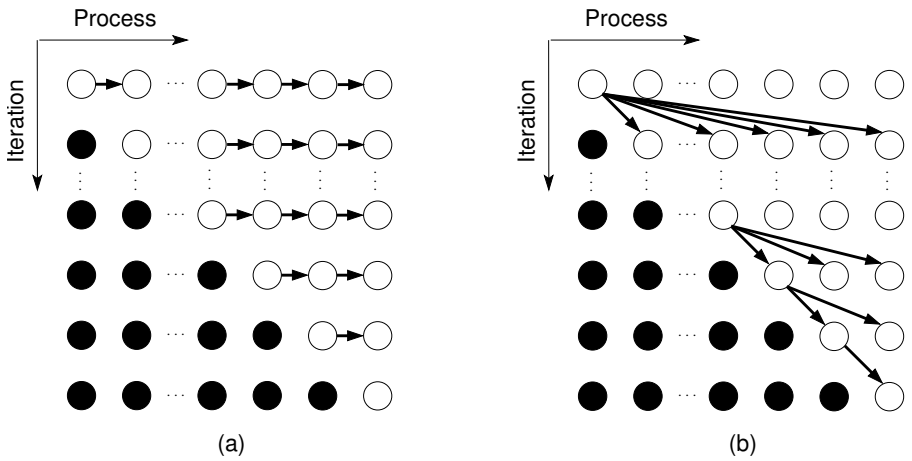


Figure 4.6: Communication structures for the parallel Gaussian elimination algorithm: (a) pipeline, and (b) broadcast. For each of the iterations, black ellipses represent processes that have completed their iterations. Thick arrows represent data flow.

```
1  void gaussian_pipeline ( float A[][], int n ) {
       int i, j, last := bc_size − 1, r := bc_rank, y := n + 1;
3      bc_plist_t *sink_pl := NULL, *src_pl := NULL;
       bc_chan_t *sink := NULL, *src := NULL;
5      /* Create Pipeline communication structure. */
       if ( r ≠ 0 ) {
7          src_pl := bc_plist_create ( 1, r − 1 );
           src := bc_src_create ( src_pl, bc_float, BC_ROLE_PIPE );
9      }
       if ( r ≠ last ) {
11         sink_pl := bc_plist_create ( 1, r + 1 );
           sink := bc_sink_create ( sink_pl, bc_float, y + 1, BC_ROLE_PIPE );
13     }
       /* Eliminate all the preceding equations. */
15     for ( i := 0; i < r; i++ ) {
           /* Receive preceding equations for elimination. */
17         bc_get ( src, &A[i][0], y + 1 );
           for ( j := i + 1; j < n; j++ )
19             A[r][j] := A[r][j] − A[r][i] * A[i][j];
           A[r][n] := A[r][n] − A[r][i] * A[i][y];
21         A[r][i] := 0;
           /* Pass on received equations. */
23         if ( r ≠ last ) bc_put ( sink, &A[i][0], y + 1 );
       }
25     /* Preceding equations have been eliminated, divide my equation. */
       for ( j := r + 1; j < n; j++ )
27         A[r][j] := A[r][j] / A[r][r];
       A[r][y] := A[r][n] / A[r][r];
29     A[r][r] := 1;
       /* Send my equation for elimination in the succeeding stages. */
31     if ( r ≠ last ) bc_put ( sink, &A[r][0], y + 1 );
       /* Destroy communication structure. */
33     if ( r ≠ 0 ) { bc_chan_destroy ( src ); bc_plist_destroy ( src_pl ); }
       if ( r ≠ last ) { bc_chan_destroy ( sink ); bc_plist_destroy ( sink_pl ); }
35 }
```

Figure 4.7: Implementation of the parallel Gaussian elimination algorithm using a pipeline communication structure. A is a consolidated matrix representing **Ax=b**; and n gives the number of linear equations in the system being solved.

The Gaussian elimination algorithm reduces a system of linear equations given by $\mathbf{Ax} = \mathbf{b}$, to an upper triangular form $\mathbf{Ux} = \mathbf{c}$. Once a system is reduced to this form, *back-substitution* [86, 49] can be applied to give the values of x_i.

A sequential implementation of the Gaussian elimination algorithm for reducing a system of linear equations to an upper triangular form is shown in Figure 4.5. This implementation can be parallelised in the outermost loop (line 3) in two ways by using the communication structures shown in Figure 4.6.

4.3.1.1 *Pipeline communication structure*

This section describes a parallel implementation of the Gaussian elimination algorithm using a pipeline communication structure. With a pipeline communication structure (Figure 4.6.a), every process participating in the computation represents a pipeline stage.

Assume that n processes are used for solving a system of n linear equations, where $\mathcal{P}_i$ is assigned equation i. The algorithm executed by each stage of the pipeline can then be expressed as follows,

```
for 1 to rank do
    Receive equation x from the preceding stage.
    Eliminate equation x from local equation.
    Send equation x to succeeding stage.
end for
Reduce local equation to triangular form.
Send local equation to succeeding stage.
```

In the above algorithm, $rank = i$ gives the rank of $\mathcal{P}_i$ in the process ensemble, assuming stage i is executed by $\mathcal{P}_i$. In the first part of the algorithm, each stage eliminates all the preceding linear equations. This is done in the for loop, where a linear equation is received from the preceding stage, which is then eliminated from the equation assigned to that stage before sending the unmodified received equation to the succeeding stage.

In the second part, after all the preceding linear equations have been eliminated, the linear equation assigned to stage i is divided by $A(i,i)$ which reduces the equation to the upper triangular form. This equation is then sent to the succeeding stage so that it can be eliminated by the succeeding stages of the pipeline.

The corresponding implementation with β-channels interfaces is shown in Figure 4.7. Because of the pipeline communication structure, each stage behaves as the consumer for the preceding stage, and producer for the succeeding stage. Two β-channels are therefore required on each stage for these two producer-consumer

roles. They are created during the communication structuring phase (lines 6–13). The source β-channel is created with a process list containing the preceding stage process (bc_rank − 1), and is given the BC_ROLE_PIPE role because the process is communicating with the preceding process only. Similarly, the sink β-channel is created with a process list containing the succeeding stage process (bc_rank + 1). This β-channel is also given the BC_ROLE_PIPE role because it is communicating with the succeeding process only. For the sink β-channel, the size of the message buffer is specified as (y + 1) so that a row in $\mathbf{Ax} = \mathbf{b}$ can be communicated. Finally, both β-channels are specialised with the bc_float data type: assuming that the variables, coefficients and constants are float values.

During the activation phase, the communication structure created in the previous phase is activated for transferring data (lines 17, 23 and 31). After a stage concludes computation, the communication structure is destroyed by destroying the β-channels and their corresponding process lists.

To relate the abstraction model to the programming model, we shall refer to the concepts discussed in Chapter 3 by using this example. As we can see from Figure 4.6, the dependency points lie within the main loop (see line 3 in Figure 4.5), which, in the parallel implementation with pipeline communication pattern (see Figure 4.7) is shown at lines 17, 23 and 31. These dependency points represent segments within the application program where data is communicated with remote processes. From these dependency points, we derive the dependency classes, which, in this case only contain one dependency point since the process communicates with either the predecessor or the successor. Furthermore, the predecessor and the successor form the projections of these dependency points, for which we create the process lists (see lines 7 and 11). We combine these projections with the appropriate role, data type and buffer size to create the encapsulating β-channel data structures (see lines 8 and 12). As discussed previously, we can now activate these β-channels to perform communications.

4.3.1.2 *Broadcast communication structure*

This section describes a parallel implementation of the Gaussian elimination algorithm using a broadcast communication structure (Figure 4.6.b). In this implementation, every process participating in the computation successively acts as the *root* of a broadcast during the iterations.

Again, assuming that n processes are utilised for solving a system of n linear equations, where $\mathcal{P}_i$ is assigned equation i, the algorithm executed by every process can be expressed as follows,

```
1  void gaussian_replication ( float A[][], int n ) {
     int i, j, last := bc_size − 1, r := bc_rank, y := n + 1;
3    bc_plist_t *sink_pl := NULL, *src_pl := NULL;
     bc_chan_t *sink := NULL, *src := NULL;
5    /* Eliminate all the preceding equations. */
     for ( i := 0; i < r; i++ ) {
7      /* Create the communication structure for receiving equations. */
       src_pl := bc_plist_create ( 1, i );
9      src := bc_src_create ( src_pl, bc_float, BC_ROLE_PIPE );
       /* Receive preceding equations for elimination. */
11     bc_get ( src, &A[i][0], y + 1 );
       for ( j := i + 1; j < n; j++ )
13       A[r][j] := A[r][j] − A[r][i] * A[i][j];
       A[r][n] := A[r][n] − A[r][i] * A[i][y];
15     A[r][i] := 0;
       /* Destroy the communication structure for receiving equations. */
17     bc_chan_destroy ( src ); bc_plist_destroy ( src_pl );
     }
19   /* Preceding equations have been eliminated, divide my equation. */
     for ( j := r + 1; j < n; j++ )
21     A[r][j] := A[r][j] / A[r][r];
     A[r][y] := A[r][n] / A[r][r];
23   A[r][r] := 1;
     /* Create communication structure for broadcasting. */
25   if ( bc_rank ≠ last )
       sink := bc_sink_create ( bc_plist_succ, bc_float,
27                 y + 1, BC_ROLE_REPLICATE );
     /* Broadcast my equation for elimination in the succeeding processes. */
29   if ( r ≠ last ) bc_put ( sink, &A[r][0], y + 1 );
     /* Destroy communication structure. */
31   if ( r ≠ last ) bc_chan_destroy ( sink );
   }
```

Figure 4.8: Implementation of the parallel Gaussian elimination algorithm using a broadcast communication structure. A is a consolidated matrix representing **Ax=b**; and n gives the number of linear equations in the system being solved.

for 1 to rank **do**
  Receive equation x from the current root.
  Eliminate equation x from local equation.
end for
Reduce local equation to triangular form.
Broadcast local equation to all the succeeding processes.

Similar to the implementation with a pipeline communication structure, all the preceding linear equations are eliminated during the first part of the algorithm. This is done within the for loop where a linear equation is received from the current root—which is $\mathcal{P}_i$ for iteration i. However, received equations are not sent to any process because the broadcast renders it unnecessary.

In the second part, after all the preceding linear equations have been eliminated, the linear equation assigned to stage i is divided by $A(i,i)$ reducing the equation to the upper triangular form. This equation is then broadcast to all the succeeding processes for elimination in those processes.

The corresponding implementation with β-channels is shown in Figure 4.8. Since the root of the broadcast changes with every iteration, the source β-channel for receiving a linear equation from the root is created dynamically within the loop (lines 8–9). This β-channel is given the BC_ROLE_PIPE role because it only communicates with the root of the broadcast. Assuming that the variables, coefficients and constants are of float data type, this β-channel is specialised with the bc_float data type. During each iteration, the preceding equation is received through this β-channel. After eliminating the received equation, the β-channel and its corresponding process list is destroyed within the loop.

After eliminating all the preceding linear equations, the process creates a sink β-channel for broadcasting its linear equation (lines 25–27). As the process is currently the root, this β-channel is given the BC_ROLE_REPLICATE role to specify the communications with all of the succeeding processes. Instead of creating a new process list, the builtin process list bc_plist_succ is used to specify all the succeeding processes. Similar to the source β-channel, this β-channel is also specialised with the bc_float data type and a request for a message buffer of y + 1 buffering units is made. Once the sink β-channel has been created, it is activated for broadcasting the linear equation (line 29). Subsequently, the sink β-channel is destroyed.

4.3.2 Fast Fourier transform

The discrete Fourier transform (DFT) has many scientific applications, such as digital signal processing, and solving partial differential equations. In this section, we

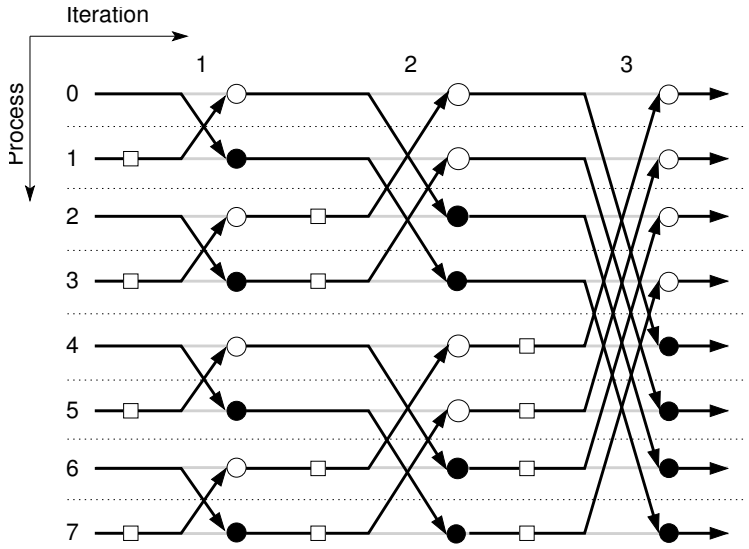


Figure 4.9: Communication structure for an 8 point radix-2 DIT fast Fourier transform. ● and ○ respectively represent subtraction and addition of values received from *partner*. □ represents multiplication by *twiddle factor*.

discuss implementation of the fast Fourier transform (FFT) algorithm due to Cooley and Tukey [34], which reduces the time complexity of computing a DFT of an n point series from $\Theta(n^2)$ to $\Theta(n \log n)$ [86, 49].

Definition 4.3.4 (Discrete Fourier transform)
Given a sequence $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ of length n, the *discrete Fourier transform* of X is defined as the sequence $Y = \langle y_0, y_1, \ldots, y_{n-1} \rangle$, where

$$y_j = \sum_{k=0}^{n-1} x_k \omega^{jk},\ 0 \leqslant j < n. \tag{4.1}$$

$\omega = e^{-2\pi i/n}$ is the principal n^{th} root of unity in the complex plane. $i = \sqrt{-1}$ and e is the base of natural logarithms. ▌

The FFT is a *divide and conquer* algorithm which recursively breaks down a DFT of the sequence X of size n into DFTs of two sequences of sizes n_1 and n_2, where $n = n_1 n_2$; along with $O(n)$ multiplications by powers of ω, often referred to as *twiddle factors*.

The simplest and most common form of the FFT is the radix-2 decimation-in-time (DIT) algorithm which decimates the problem size by a factor of 2. In this section, we implement this algorithm with β-channels. Assuming that $n = 2^k$ for some constant k, the *Butterfly* communication structure required by a one-dimensional 8 point radix-2 DIT algorithm is shown in Figure 4.9. In order to fully understand the

```
1  void fourier ( int ncoeff, complex_t *coeff ) {
     int i, j, step, iter, nbit, nbc, pbit, *partner, sign;
3    bc_chan_t **src := NULL, **sink := NULL; /* Source and sink beta-channels. */
     bc_plist_t **plists; /* Set of process lists for each iteration. */
5    bc_dtype_t *ntype; /* Custom data type. */
     complex_t recv; /* Temporary variable for receiving coefficient. */
7    iter := ( int ) log2 ( ( double) ncoeff ); /* Number of iterations. */
     pbit := iter; /* Number of bits in process rank. */
9    nbc := iter; /* Number of beta-channels. */
     /* Create custom data type. */
11   ntype := bc_dtype_create ( sizeof ( complex_t ) );
     /* Allocate memory for the communication structure. */
13   partner := ( int * ) malloc ( sizeof ( int ) * nbc );
     plists := ( bc_plist_t ** ) malloc ( sizeof ( bc_plist_t * ) * nbc );
15   src := ( bc_chan_t ** ) malloc ( sizeof ( bc_chan_t * ) * nbc );
     sink := ( bc_chan_t ** ) malloc ( sizeof ( bc_chan_t * ) * nbc );
17   /* Create Butterfly communication structure. */
     for ( i := iter, j := 0; i; i--, j++ ) {
19     nbit := iter - i; partner[j] := bit_complement ( idx, nbit );
       plists[j] := bc_plist_create ( 1, partner[j] );
21     src[j] := bc_src_create ( plists[j], ntype, BC_ROLE_PIPE );
       sink[j] := bc_sink_create ( plists[j], ntype, 1, BC_ROLE_PIPE );
23   }
     /* Execute communication structure. */
25   for ( i := 1, j := 0, step := 0; i ≤ iter; i++, j++ ) {
       if ( partner[j] < bc_rank ) {
27       multiply_twiddle ( coeff, 1 ≪ i, step); /* Multiply with twiddle factor. */
         sign := 0; step += i;
29     } else sign := 1;
       /* Exchange complex coefficients. */
31     bc_put ( sink[j], coeff, 1 );
       bc_get ( src[j], &recv, 1 );
33     if ( sign > 0 ) complex_addition ( coeff, &recv );
       else complex_subtraction ( coeff, &recv);
35   }
     /* Destroy communication structure. */
37   for ( i := 0; i < nbc; i++ ) {
       bc_chan_destroy ( src[i] ); bc_chan_destroy ( sink[i] );
39     bc_plist_destroy ( plists[i] );
     }
41   bc_dtype_destroy ( ntype ); /* Destroy custom data type. */
     free ( src ); free ( sink ); free ( plists ); free (partner); /* Deallocate memory. */
43 }
```

Figure 4.10: β-channel implementation of the radix-2 decimation-in-time fast Fourier transform algorithm. It is interesting to note that each process defines a partner for each of the iterations before commencing communication (lines 19). We can further improve this implementation by pre-calculating the *twiddle factors*.

β-channel implementation of the Butterfly communication structure, inter-process communications are enforced during all iterations by assigning element x_i of the sequence X to process $\mathcal{P}_i$ from the set of n available processes. For simplicity, the permutations of the elements [86, page 208] in X are omitted. The algorithm executed by every process can therefore be represented as follows,

```
p ← 0; j ← 0; iter ← log2 n;
for i ← 1 to iter do
   if partnerj < rank then
      coef ← coef × ω_{2^i}^p
      p ← p + i
      sign ← 0;
   else
      sign ← 1;
   end if
   Send my coefficient coef to partnerj;
   Receive coefficient recv from partnerj;
   if sign = 1 then
      coef ← recv + coef;
   else
      coef ← recv − coef;
   end if
   j ← j + 1;
end for
```

The coef gives the value of x_i in X, which was assigned to $\mathcal{P}_i$ at the start of the execution. Within the for loop, the value of $\mathrm{partner}_j$ gives the remote process with which the process should communicate during iteration i. rank gives the rank of the executing process.

If $\mathrm{partner}_j < \mathrm{rank}$, the coefficient coef on $\mathcal{P}_{\mathrm{rank}}$ is multiplied[2] by ω_q^p. This coefficient is then exchanged with the coefficient on process $\mathrm{partner}_j$, receiving the new coefficient in recv. After the exchange, coef is subtracted, or added, to recv to give the new value of coef based on the following conditions,

$$\mathrm{coef} = \begin{cases} \mathrm{recv} - \mathrm{coef} & \text{if } \mathrm{partner}_j < \mathrm{rank}, \\ \mathrm{recv} + \mathrm{coef} & \text{otherwise.} \end{cases}$$

The β-channel implementation of the radix-2 DIT algorithm is shown in Figure 4.10. The implementation has two phases: the communication structuring phase and the communication activation phase. Before structuring the communications, a custom

[2]For simplicity, the value of ω_q^p is calculated for every iteration; however, in practice, pre-calculating these values would make for better performance.

data type is first created (line 11). This is necessary because the built-in data types defined by the library (see Section 4.2) do not provide complex data types required for exchanging the complex coefficients.

Based on the butterfly communication structure, the only communications required by the algorithm at each of the $\log_2 n$ iterations are exchanges of coefficients between two processes. Therefore, each process creates a pair of source and sink β-channels for receiving and sending coefficients (lines 18–23). Even though the *partner* process changes with every iteration (line 19), all the β-channels are created *a priori* instead of creating them dynamically for each iteration. Both source and sink β-channels share a process list which specifies *partner* (line 20). Finally, the BC_ROLE_PIPE role is specified for both β-channels.

Once the butterfly communication structure has been created, they are activated during the iterations to exchange coefficients (lines 31–32). After completing all the $\log_2 n$ iterations, the butterfly communication structure and the custom data type are destroyed (lines 37–41).

4.3.3 Odd-even transposition sorting

Sorting is one of the most common activities in data processing. By sorting a set of data, future references to that data set can be performed more efficiently. In fact, sorting data forms a crucial part of most parallel algorithms. In this section, we implement the odd-even transposition parallel sorting algorithm by using β-channels.[3]

Let us assume that $S = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence of data to be sorted with n processes. For simplicity, also assume that a_i is assigned to $\mathcal{P}_i$. The odd-even transposition sorting algorithm performs $n/2$ iterations, which have two phases: (1) even exchange and (2) odd exchange. In the first phase, all even ranked processes compare their values with the values in the succeeding process. If necessary, the values are exchanged so that the lower ranked process gets the smaller value. Similarly, in the second phase, the values in every odd ranked process are compared with the values in the succeeding processes. If necessary, the values are exchanged so that the lower ranked process gets the smaller value. After $n/2$ iterations, the values stored in the rank ordered processes give the required sorted values.

The algorithm executed by all the n participating processes can be expressed as,

```
if (rank mod 2) ≠ 0 then
    odd ← rank − 1; even ← rank + 1;
```

[3] Although the odd-even transposition sorting algorithm is not the most efficient sorting algorithm that is available, we have chosen this algorithm in order to illustrate implementation of the unique communication pattern it manifests.

```
1  int local_upper, recv_upper, *recv_base;
   size_t bytes;
3  void oddeven (int nlocal, int *elem) {
     bc_plist_t *odd_pl := NULL, *even_pl := NULL;
5    bc_chan_t *odd_src, *odd_sink, *even_src, *even_sink;
     int odd_rank, even_rank, *workspace, iter, i;
7    workspace := ( int * ) calloc ( nlocal ≪ 1, sizeof ( int ) );
     bytes := nlocal * sizeof ( int ); local_upper := nlocal − 1;
9    recv_upper := ( nlocal ≪ 1 ) − 1; recv_base := workspace + nlocal;
     iter := bc_size / (2 * nlocal);
11   /* Create exchange communication structure. */
     if ( bc_rank % 2 ) { odd_rank := bc_rank − 1; even_rank := bc_rank + 1; }
13   else { odd_rank := bc_rank + 1; even_rank := bc_rank − 1; }
     if ( odd_rank > −1 ∧ odd_rank < bc_size ) {
15     odd_pl := bc_plist_create ( 1, odd_rank );
       odd_src := bc_src_create ( odd_pl, bc_int, BC_ROLE_PIPE );
17     odd_sink := bc_sink_create ( odd_pl, bc_int, nlocal, BC_ROLE_PIPE );
     }
19   if ( even_rank > −1 ∧ even_rank < bc_size ) {
       even_pl := bc_plist_create ( 1, even_rank );
21     even_src := bc_src_create ( even_pl, bc_int, BC_ROLE_PIPE );
       even_sink := bc_sink_create ( even_pl, bc_int, nlocal, BC_ROLE_PIPE );
23   }
     qsort ( elem, nlocal, sizeof ( int ), compare ); /* Sort local elements. */
25   /* Odd-even transposition exchange loop. */
     for ( i := 0; i < iter; i++ ) {
27     if ( even_pl ≠ NULL ) {
         bc_put ( even_sink, elements, nlocal ); bc_get ( even_src, recv_base, nlocal );
29       compare_exchange ( nlocal, elements, workspace, bc_rank < even_rank );
       }
31     if ( odd_pl ≠ NULL ) {
         bc_put ( odd_sink, elements, nlocal ); bc_get ( odd_src, recv_base, nlocal );
33       compare_exchange ( nlocal, elements, workspace, bc_rank < odd_rank );
       }
35   }
     /* Destroy communication structure. */
37   if ( odd_pl ≠ NULL ) {
       bc_chan_destroy (odd_src); bc_chan_destroy (odd_sink);
39     bc_plist_destroy (odd_pl);
     }
41   if ( even_pl ≠ NULL ) {
       bc_chan_destroy (even_src); bc_chan_destroy (even_sink);
43     bc_plist_destroy (even_pl);
     }
45   free ( workspace ); /* Deallocate workspace. */
   }
```

Figure 4.11: β-channel implementation of the odd-even transposition sorting algorithm. While creating the exchange communication structure (lines 12–23), each process creates four β-channels—two source and sink β-channel pairs—for communicating data with a partner in each of the iterations (line 26–35).

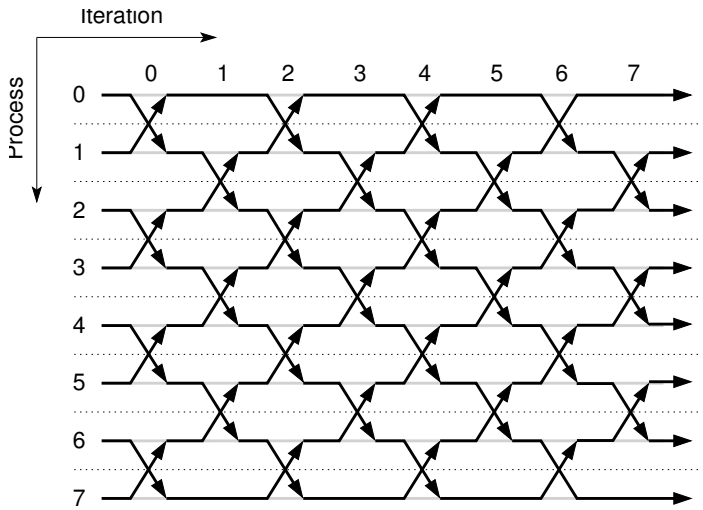


Figure 4.12: Communication structure for an 8 point odd-even transposition sort. The pattern requires only two pairs of source and sink β-channels: each pair communicating with a partner in one of the odd, or even, phases.

```
else
   odd ← rank + 1; even ← rank − 1;
end if
for i ← 1 to n/2 do
   if (even > −1) ∧ (even < n) then
      put (even, a_j); get (even, t);
      if rank < even then
         a_j ← minimum (t, a_j);
      else
         a_j ← maximum (t, a_j);
      end if
   end if
   if (odd > −1) ∧ (odd < n) then
      put (odd, a_j); get (odd, t);
      if rank < odd then
         a_j ← minimum (t, a_j);
      else
         a_j ← maximum (t, a_j);
      end if
   end if
end for
```

Here, rank gives the rank of the executing process. even and odd respectively refer to the processes with which the executing process communicates during the even or odd exchange phases. t stores the received value during comparisons.

The communication structure manifested by the odd-even transposition algo-

rithm for $n = 8$ is shown in Figure 4.12. If we consider any process, say $\mathcal{P}_1$, we observe that two β-channels are required for the odd exchange phase, and another two for the even exchange phase. Hence, in the β-channel implementation (see Figure 4.11), the communication structure is defined by four β-channels: odd_src, odd_sink, even_src, and even_sink.

The β-channels are created (lines 12–23) with BC_ROLE_PIPE role (similar to the implementation of the fast Fourier transformation in Section 4.3.2) for exchanging values. This implementation allows more than one value to be assigned to a single process; however, these values should be sorted before entering the iteration (line 24). The number of values assigned to a process is given by nlocal, and the number of transposition iterations is given by $n/(2 * \text{nlocal})$ (line 10). In order to improve performance by avoiding memory copy during comparisons, we allocate a working memory workspace which is used for both receiving and comparing data.

The function compare_exchange()[4] compares the values received in recv (which is a pointer within the working memory workspace) with its local value elem, storing the minimum value in elem if the rank of the process is less than the rank of the remote process; the maximum value is stored otherwise (line 29 and 33). Once all of the iterations have been executed, the communication structure is destroyed by deallocating the four β-channels and corresponding process lists (lines 37–44).

4.3.4 Mandelbrot set task farm

The Mandelbrot set is a *fractal* defined by a set of points c in the complex plane for which the iteratively defined sequence

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

does not tend to *infinity*. After reformulating in terms of the real and imaginary parts of the complex plane coordinates X and Y, we get

$$\begin{aligned} x_{n+1} &= x_n^2 - y_n^2 + a \\ y_{n+1} &= 2x_n y_n + b, \quad \text{where } c = a + ib. \end{aligned}$$

It can be shown that the sequence will tend to infinity, and c will be outside the Mandelbrot set if $|z_n| > 2$, where $|z_n| = \sqrt{x_n^2 + y_n^2}$ is the *modulus* of z_n [28, page 124]. This value, often referred to as the *bail-out* value, allows termination of the iteration for points outside the Mandelbrot set. For points inside the Mandelbrot

[4] Auxillary functions can be found in Appendix A.

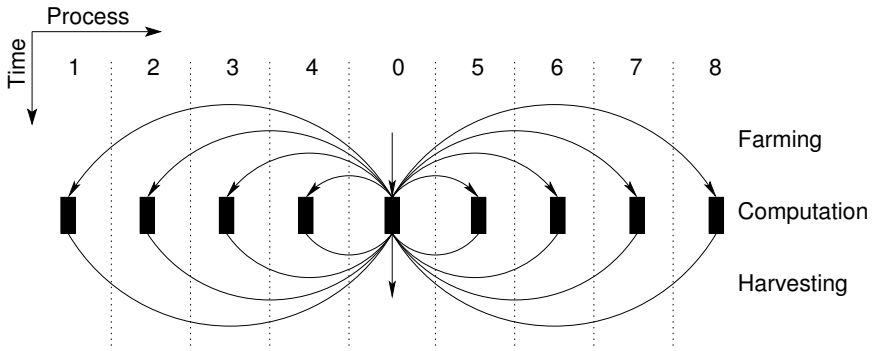


Figure 4.13: Communication structure for the Mandelbrot set task farm with one farmer and 8 worker processes. Blocks of complex points are sent to each of the worker processes, which convert the points to colour codes. The codes are harvested by the farmer process to generate a graphical representation of the complex points which belong in the Mandelbrot set.

set the loop never terminates. This infinite loop should therefore be terminated after a pre-determined number of iterations, max_iter. The iterative sequential algorithm for finding out if a complex point belongs in the Mandelbrot set is given below (adapted from [83, page 191]).

$r \leftarrow i \leftarrow j \leftarrow 0;$
while $j < max\_iter$ **do**
  $t \leftarrow r^2 - i^2 + x;$
  $i \leftarrow 2 \times r \times i + y;$
  $r \leftarrow t;$
  $a \leftarrow r^2 + i^2;$
  if $a > 4$ **then**
    break;
  end if
  $j \leftarrow j + 1;$
end while
return j;

To parallelise the algorithm for a set of n^2 complex points, *data partitioning* is used. Each process applies the above algorithm to each of the complex points in the block which was received from the farmer process. If we employ n processes, the n^2 complex points can be partitioned into n blocks, each containing n complex points. The data partitioning can be done row-major or column-major. For the purpose of this discussion, we choose row-major distribution of data: meaning, $\mathcal{P}_i$ gets the i^{th} row containing n complex points given by (x_i, y_j) for all $0 \leqslant j < n$.

```
1  void farmer ( int rows, int cols, float rstart, float rend, float istart, float iend ) {
       bc_chan_t *sink, *src;
3      bc_dtype_t *ntype;
       complex_t *input, *iptr;
5      int *result, *optr, i, j, offset, iter, npoints, nworkers;
       float rrange, irange, real, img, rstep, istep;
7      npoints := rows * cols; nworkers := bc_size - 1;
       rrange := rend - rstart; irange := iend - istart;
9      /* Allocate memory for complex points and the results. */
       input := ( complex_t * ) calloc ( npoints, sizeof ( complex_t ) );
11     results := ( int * ) calloc ( npoints, sizeof ( int ) );
       /* Create custom data type. */
13     ntype := bc_dtype_create ( sizeof ( complex_t ) );
       /* Create Farm communication structure. */
15     src := bc_src_create ( bc_plist_xall, bc_int, BC_ROLE_COLLECT );
       sink := bc_sink_create ( bc_plist_xall, ntype, cols, BC_ROLE_SPREAD );
17     /* Generate the coordinates. */
       img := istart; rstep := rrange / cols; istep := irange / rows;
19     for ( i := 0; i < rows; i++, img += istep ) {
           real := rstart;
21         for ( j := 0; j < cols; j++, real += rstep ) {
               iptr := ( input + cols * i + j );
23             iptr→real := real; iptr→img := img;
           }
25     }
       /* Farm the coordinates and harvest colour codes. */
27     iptr := plane; optr := results;
       offset := nworkers * cols;
29     iter := ceil ( rows / nworkers );
       for ( i := 0; i < iter; i++ ) {
31         bc_put ( sink, iptr, cols ); /* Farm blocks of complex points. */
           bc_get ( src, optr, cols ); /* Harvest results (colour codes). */
33         p_ptr += offset; optr += offset;
       }
35     /* Destroy communication structure. */
       bc_chan_destroy ( sink ); bc_chan_destroy ( src );
37     bc_dtype_destroy ( ntype ); /* Destroy custom data type. */
       generate_image ( results, rows, cols ); /* Generate image file. */
39     free ( input ); free ( results );
   }
```

Figure 4.14: β-channel implementation of the Mandelbrot set farmer function with deterministic roles, based on the uniform distribution of complex points. The n^2 complex points are partitioned row-wise so that each row is mapped to one of the worker process. This is defined statically, and therefore prevents faster worker processes from computing more complex points than the slower ones.

```
1  void worker ( int rows, int cols ) {
       bc_chan_t *src, *sink;
3      bc_plist_t *farmer;
       bc_dtype_t *ntype;
5      complex_t *input;
       int *result, i, iter;
7      /* Allocate working memory. */
       input := ( complex_t * ) calloc ( cols, sizeof ( complex_t ) );
9      results := ( int * ) calloc ( cols, sizeof ( int ) );
       /* Create custom data type. */
11     ntype := bc_dtype_create ( sizeof ( complex_t ) );
       /* Create the communication structure. */
13     farmer := bc_plist_create ( 1, 0 ); /* Process 0 is farmer. */
       src := bc_src_create ( farmer, ntype, BC_ROLE_PIPE );
15     sink := bc_sink_create ( farmer, bc_int, cols, BC_ROLE_PIPE );
       /* Get complex points and calculate. */
17     iter := ceil ( rows / ( bc_size - 1 ) );
       for ( i := 0; i < iter; i++ ) {
19         bc_get ( src, row, cols ); /* Get complex points. */
           process_mandel ( cols, input, results ); /* Process complex points. */
21         bc_put ( sink, result, cols ); /* Send results (colour codes). */
       }
23     /* Destroy communication structure. */
       bc_chan_destroy ( src ); bc_chan_destroy ( sink );
25     bc_plist_destroy( farmer );
       bc_dtype_destroy ( ntype );
27     free ( input ); free ( results );
   }
```

Figure 4.15: β-channel implementation of the Mandelbrot set worker function, based on the uniform distribution of complex points. A process retrieves the block of complex points assigned to it by the farmer. Once the complex points in that block have been converted to colour codes, they are returned to the farmer. This continues until all the rows assigned to the process are exhausted.

Uniform distribution of data

Assuming that the data are distributed and collected by a farmer process $\mathcal{P}_n$ which is not in the set of n worker processes, the communication structure manifested by the Mandelbrot set task farm is shown in Figure 4.13. Based on this communication structure, the algorithm executed by the *farmer* process can be represented as:

```
p ← points;
r ← results;
b ← n × ncols;
iter ← ⌈nrows/n⌉;
for i ← 1 to iter do
   spread (workers, p, ncols);
   collect(workers, r, ncols);
   p ← p + b;
   r ← p + b;
end for
```

After a row of *ncols* complex points is received from the *farmer*, the *worker* executes the following algorithm to process and return the colour codes.

```
loop
   get (farmer, &v[0], ncols);
   if no data received then
      break;
   end if
   for i ← 0 to ncols − 1 do
      r[i] ← mandelbrot(v[i].real, v[i].img);
   end for
   put (farmer, &r[0], ncols);
end loop
```

The corresponding β-channel implementation of the *farmer* and *worker* algorithms is shown in Figure 4.14, and Figure 4.15 respectively.

In this implementation of the Mandelbrot set task farm, we use uniform distribution of 'work' (the number of points per block) for each of the worker processes. We can observe this in the implementation of the farmer function Figure 4.14. While creating the β-channel for communicating with the workers, the uniform distribution of complex points is accounted for by the roles BC_ROLE_SPREAD and BC_ROLE_COLLECT (lines 15–16). In the execution loop, the farmer process sends different rows to the worker functions by spreading (line 31) the complex points initialised in lines 19–25. In line 32, the farmer collects the colour codes from the worker processes and stores them, aligning them based on the rank of the worker

```
#define PIX_ROWS 1024 /* Number of rows. */
#define PIX_COLS 1024 /* Number of complex points per row. */
typedef struct complex_s{ float real; float img; } complex_t;
typedef struct { int row; complex_t point[PIX_COLS]; } irow_t;
typedef struct { int row; int color[PIX_COLS]; } orow_t;
void farmer_refined ( float rstart, float rend, float istart, float iend  ) {
    bc_chan_t *sink, *src;
    bc_dtype_t *idtype, *odtype;
    int i, j, nworkers, dunits;
    orow_t output[PIX_ROWS];
    float rrange, irange, real, img, rstep, istep;
    rrange := rend - rstart; irange := iend - istart;
    /* Create communication structure. */
    idtype := bc_dtype_create ( sizeof ( irow_t ) );
    odtype := bc_dtype_create ( sizeof ( orow_t ) );
    nworkers := bc_size - 1; /* Number of workers. */
    duints := PIX_ROWS + nworkers + 1; /* +1 data unit required. */
    sink := bc_sink_create ( bc_plist_xall, idtype, dunits, BC_ROLE_FARM );
    src := bc_src_create ( bc_plist_xall, odtype, BC_ROLE_HARVEST );
    /* Generate and commit complex points. */
    img := istart; rstep := rrange / PIX_COLS; istep := irange / PIX_ROWS;
    for ( i := 0; i < PIX_ROWS; i++, img += istep ) {
        real := rstart;
        bc_vptr ( sink, irow_t )→row := i; /* Specify which row. */
        for ( j := 0; j < PIX_COLS; j++ ) {
            bc_vptr (sink, irow_t )→point[j].real := real;
            bc_vptr (sink, irow_t )→point[j].img := img;
            real += rstep;
        }
        bc_commit ( sink ); /* Commit this row. */
    }
    /* Commit termination values. */
    for ( i := 0, j := PIX_ROWS; i < nworkers; i++ ) {
        bc_vptr ( sink, irow_t )→row := -1; bc_commit ( sink );
    }
    /* Harvest results (colour codes). */
    for ( i := 0; i < PIX_ROWS; i++) bc_get ( src, &output[i], 1 );
    /* Destroy communication structure. */
    bc_chan_destroy ( src ); bc_chan_destroy ( sink );
    bc_dtype_destroy ( idtype ); bc_dtype_destroy ( odtype );
    generate_image ( output, PIX_ROWS, PIX_COLS ); /* Generate image file. */
}
```

Figure 4.16: β-channel implementation of the Mandelbrot set farmer function with non-deterministic roles, based on runtime determination of complex point to process mapping. The n^2 complex points are still partitioned into rows of complex points; however, which process gets a particular row is determined at runtime. This removes the restrictions imposed by the implementation with deterministic roles, and therefore increases efficiency because faster worker processes can compute more complex points than slower ones.

```
1  int worker_refined ( void ) {
       bc_chan_t *src, *sink;
3      bc_plist_t *farmer;
       bc_dtype_t *idtype, *odtype;
5      irow_t i; /* For receiving input data. */
       orow_t o; /* For computing output data. */
7      farmer := bc_plist_create ( 1, 0 ); /* Create the farmer process list. */
       /* Create communication structure. */
9      idtype := bc_dtype_create ( sizeof(irow_t) );
       odtype := bc_dtype_create ( sizeof(orow_t) );
11     src := bc_src_create ( farmer, idtype, BC_ROLE_PIPE );
       sink := bc_sink_create ( farmer, odtype, 2, BC_ROLE_PIPE );
13     /* Continue computations while unprocessed data are available. */
       while (1) {
15         bc_get ( src, &i, 1 ); /* Get complex points. */
           if ( i.row = -1 ) break; /* No more unprocessed data. */
17         process_mandel ( PIX_COLS, i.point, o.color ); /* Process complex points. */
           o.row := i.row; /* The row that was processed. */
19         bc_put ( sink, &o, 1 ); /* Send results (colour codes). */
       }
21     /* Destroy communication structure. */
       bc_chan_destroy ( src ); bc_chan_destroy ( sink );
23     bc_dtype_destroy ( idtype );bc_dtype_destroy ( odtype );
       bc_plist_destroy ( farmer );
25     return 0;
   }
```

Figure 4.17: β-channel implementation of the Mandelbrot set worker function, based on non-deterministic distribution of complex points. Workers continue processing rows of complex points until all the rows on the farmer have exhausted. The number of iterations that a worker is permitted to execute is not predefined. This implementation is therefore more efficient than the deterministic version shown in Figure 4.15.

process which sent the block. Finally, once all the complex points have been processed, the colour codes are converted into a graphical representation (for example, an image file), as shown in line 38.

Non-deterministic distribution of data

As one can observe, the uniform distribution of data is not efficient because of the associated 'determinism' which prevents a faster process from processing more complex points than the slower ones. In order to improve this situation, we shall now implement the Mandelbrot set task farm with non-deterministic roles: BC_ROLE_FARM and BC_ROLE_HARVEST.

When non-deterministic roles are used, the data distribution cannot be predefined statically. In principle, therefore, any process can compute any complex point (or a row of complex points, in case we decrease the granularity of the partition). Which process gets which complex point (or row) is determined at runtime, allowing faster processes to compute more complex points. Most of the implementation details, with exception to the non-determinism, are similar to the ones shown in Figure 4.14, and Figure 4.15. The differences are as follows. Firstly, each row (if we consider reduced granularity) which is sent to a worker process should be tagged with the row index. If this is not done, the farmer process does not have enough information to align the colour codes received from a worker process since the mapping of row to worker is non-deterministic. Secondly, because a worker can continue processing complex points for as long as unprocessed complex points are available, we must define a condition which marks the point when the processing should end.

The β-channel implementations which incorporate these differences are shown in Figure 4.16 and Figure 4.17. At lines 18–19 of the farmer function, the source and sink β-channels are created, respectively with the roles BC_ROLE_HARVEST and BC_ROLE_FARM. Each row is tagged with the row index while initialising the complex points (line 24). During initialisation of the complex points (lines 25–29), instead of using memory copy interface, bc_put(), we use the alternative interfaces bc_vptr() and bc_commit(), which removes the need for intermediate memory copy. The complex points for a row are therefore set directly into the buffer of the sink β-channel. Once the values have been initialised, they are committed to the buffer (line 30). To address the condition for termination, n dummy termination values are committed to the sink β-channel (line 33–35).

On the worker function, instead of executing a pre-defined number of iterations, the process executes an infinite loop (line 14–20). The termination condition is checked after receiving a row of complex points (line 16). Once all of the complex

```
1  void matrix_sequential (int A[][], int b[][], int c[][], int l, int m, int n) {
       int i, j, k;
3      for (i := 0; i < l; i++)
           for (j := 0; j < n; j++) {
5              c[i][j] := 0;
               for (k := 0; k < m; k++)
7                  c[i][j] += A[i][k] * b[k][j];
           }
9  }
```

Figure 4.18: Sequential implementation of the matrix multiplication algorithm. This can be parallelised by using a block-oriented parallel algorithm, which divides the matrices into sub-matrices, so that the product of sub-matrices can be computed simultaneously.

points in the row have been processed, the colour codes are returned to the farmer by tagging them with the row index received with the row of complex points (line 18).

4.3.5 Matrix multiplication

In this section, we discuss β-channel implementation of the block-oriented parallel matrix multiplication algorithm [86].

Definition 4.3.5 (**Matrix multiplication**)
The product of an $l \times m$ matrix **A** and an $m \times n$ matrix **B** is an $l \times n$ matrix **C** where the elements of **C** are calculated as follows,

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}, \quad 0 \leqslant i < l \text{ and } 0 \leqslant j < n.$$

Here, a_{ij} gives the element on row i and column j of matrix **A** for $0 \leqslant i < l$ and $0 \leqslant j < m$. Similarly, b_{ij} gives the element on row i and column j of matrix **B** for $0 \leqslant i < m$ and $0 \leqslant j < n$. ∎

The sequential implementation of the matrix multiplication algorithm [49] is shown in Figure 4.18. This sequential implementation requires execution time $O(n^3)$ considering that the computation (line 7) takes unit time. The performance can be improved by parallelising the implementation using a block-oriented algorithm.

First, assume that l and n are multiples of p (the number of processes available for the computation). Then partition the two matrices **A** and **B** into row and column blocks as shown in Figure 4.19.a, assuming 4 processes are available for the computation. Then, assign the first row block to $\mathcal{P}_0$, second row block to $\mathcal{P}_1$, and so on. Similarly, assign the first column block to $\mathcal{P}_0$, second column block to $\mathcal{P}_1$, and

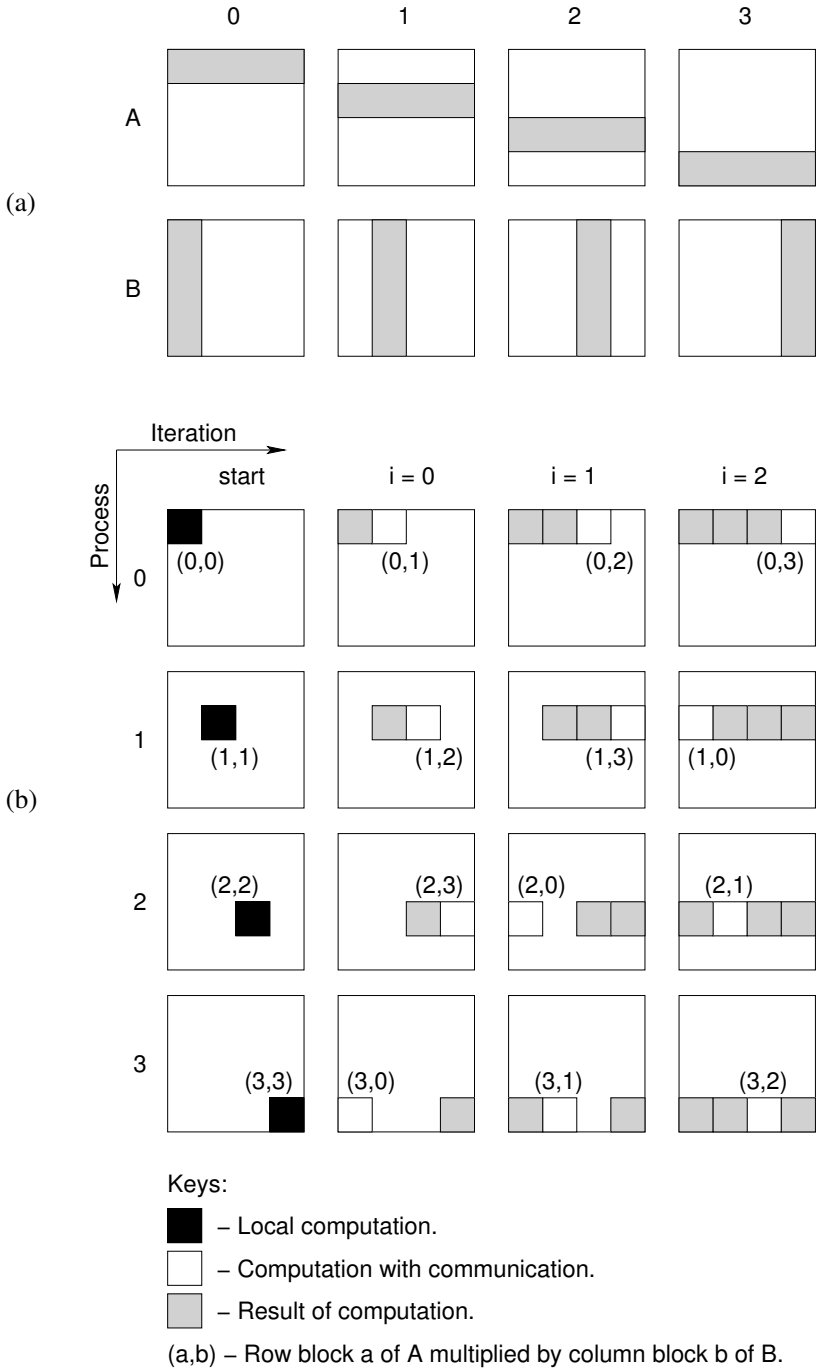


Figure 4.19: (a) Initial distribution of matrix A and matrix B blocks over 4 processes, (b) Iterations of the matrix multiplication. At the beginning of the execution, every process multiplies its two local blocks without invoking any communication. After that, blocks are communicated to the succeeding process (forming a ring topology as shown in Figure 4.20.b). When the program terminates, each process finishes with a result row, which is collected to give the matrix product.

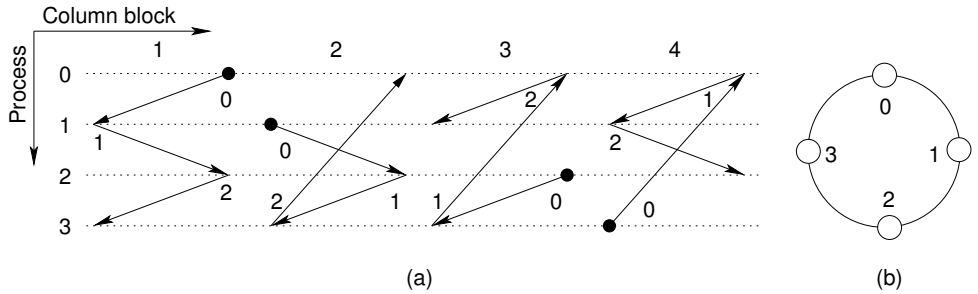


Figure 4.20: (a) Communication trace of the column blocks. • represents start of trace, arrow represents communication. The label at the tail of each arrow gives the iteration when the communication occurs. For example, column block one starts at $\mathcal{P}_0$, and travels through $\mathcal{P}_1$ and $\mathcal{P}_2$, and finally reaches $\mathcal{P}_3$. (b) A *ring topology* which arranges an ensemble of four processes to form a ring network.

so on. This constitutes the data partitioning required by the block-oriented scheme for the parallelisation.

By organising the processes to form a *Ring* topology (Figure 4.20.b), the parallelisation can be achieved as shown in Figure 4.19.b. Before commencing the iteration, a process uses its row and column blocks for performing local computation. Then, the iterations which require data communication with other processes begin. After entering the iteration, every process sends its column to its successor in the *ring*. Subsequently, it receives a new column block from its predecessor, which is multiplied by its row to give the result block. The iteration continues until all the processes have multiplied their row blocks with all of the column blocks created during the data partition. After the iterations have been completed, the resulting rows are *collected* on a root process, thus giving the final product of the two matrices. The communication trace for each column block is shown in Figure 4.20.a.

The β-channel implementation of this algorithm is shown in Figure 4.21. The implementation is divided into three parts. In the first part, the data is partitioned (lines 15–17), so that each process is assigned its corresponding row and column blocks. As the matrices are represented in *row-major form*, the *transpose* of the matrix **B** (line 16) is performed so that initialisation of column blocks can be done by direct memory copy. At the end of this part, rows and cols are initialised to the corresponding row and column block.

In the second part, after partitioning the matrices, the β-channels that correspond to the *ring topology* are created during the communication structuring phase (lines 19–23). Both source and sink β-channels are created with BC_ROLE_PIPE role as they communicate with one process only: predecessor or successor in the ring. The actual multiplications are then performed by entering the communication ac-

```
1  void matrix_multiply ( int nra, int nca, int *a, int nrb, int ncb, int *b ) {
       int nrblk, ncblk, rb_size, cb_size, last_rank, i, *rows, *cols, *result;
3      bc_plist_t *src_pl, *sink_pl;
       bc_chan_t *src, *sink;
5      nrblk := nra / bc_size; /* Number of rows per row block. */
       ncblk := ncb / bc_size; /* Number of columns per column block. */
7      rb_size := nrblk * nca; /* Size of matrix A row block. */
       cb_size := ncblk * nrb; /* Size of matrix B column block. */
9      /* Allocate working memory. */
       rows := ( int * ) calloc ( rb_size, sizeof ( int ) );
11     cols := ( int * ) calloc ( cb_size, sizeof ( int ) );
       if ( bc_rank = 0 ) result := ( int * ) calloc ( nra * ncb, sizeof ( int ) );
13     else result := ( int * ) calloc ( nrblk * ncb, sizeof ( int ) );
       /* Get my row and column blocks. */
15     memcpy ( rows, a + rb_size * bc_rank, rb_size * sizeof ( int ) );
       transpose ( nrb, ncb, b ); /* For contiguous memory copy. */
17     memcpy ( cols, b + cb_size * bc_rank, cb_size * sizeof ( int ) );
       /* Create a Ring topology. */
19     last_rank := bc_size - 1;
       src_pl := bc_plist_create ( 1, ( bc_rank = last_rank ) ? 0 : bc_rank + 1 );
21     sink_pl := bc_plist_create ( 1, ( bc_rank = 0 ) ? last_rank : bc_rank - 1 );
       src := bc_src_create ( src_pl, bc_int, BC_ROLE_PIPE );
23     sink := bc_sink_create ( sink_pl, bc_int, cb_size, BC_ROLE_PIPE );
       /* Process my row block. */
25     multiply_blocks ( result, nrblk, nca, rows, ncblk, ncb, cols, 0 );
       for ( i := 1; i < bc_size; i++ ) {
27       bc_put ( sink, cols, cb_size ); bc_get ( src, cols, cb_size );
         multiply_blocks ( result, nrblk, nca, rows, ncblk, ncb, cols, i );
29     }
       /* Destroy Ring topology. */
31     bc_chan_destroy(src); bc_chan_destroy(sink);
       bc_plist_destroy(src_pl); bc_plist_destroy(sink_pl);
33     /* Reduce partial row blocks at Process 'zero'. */
       rb_size := nrblk * ncb; /* Update row block size. */
35     if ( bc_rank = 0 ) {
         src := bc_src_create ( bc_plist_xall, bc_int, BC_ROLE_COLLECT );
37       bc_get ( src, result + rb_size, rb_size ); bc_chan_destroy(src);
       } else {
39       sink_pl := bc_plist_create ( 1, 0 );
         sink := bc_sink_create ( sink_pl, bc_int, rb_size, BC_ROLE_PIPE );
41       bc_put ( sink, result, rb_size );
         bc_chan_destroy ( sink ); bc_plist_destroy ( sink_pl );
43     }
       free ( rows ); free ( cols ); free ( result );
45 }
```

Figure 4.21: β-channel implementation of the block-oriented matrix multiplication algorithm. We first arrange the processes into a *ring topology* (lines 19–23). Each process then multiplies the row and column blocks that are locally available (line 25). When a new column block is required, it is obtained from the previous process in the ring (line 27).

tivation phase. Before entering the communication activation phase, however, the local block multiplications which do not require column communications are performed (line 25) by using the function multiply_blocks(). Each process then enters the iteration (line 26), within which old columns are sent to the successor (line 27), and new columns are received from the predecessor (line 27). Local row blocks and the received column blocks are then multiplied (line 28); storing the resulting row in result. Upon completion, the *ring topology* is destroyed by deallocating the β-channels and their corresponding process lists (lines 31–32).

In the third part, result rows from all the processes are collected on $\mathcal{P}_0$. For this, $\mathcal{P}_0$ creates a source β-channel (line 36) with BC_ROLE_COLLECT for collecting result rows from all the other processes, bc_plist_xall. However, all of the processes other than $\mathcal{P}_0$ create a sink β-channel (lines 39–40) with BC_ROLE_PIPE role as they communicate with $\mathcal{P}_0$ only. These β-channels are activated, storing the resulting product in result on $\mathcal{P}_0$. The source and sink β-channels, and corresponding process lists, if necessary, are then destroyed.

4.4 Skeletal parallel programming

In the previous sections we have discussed implementation of several non-trivial parallel algorithms by using β-channels. From these implementations, we can observe that out of the two phases of the application development exercise (see Section 4.1), the communication structuring phase is relatively more complicated than the communication activation phase, which is quite simple once the communication structures have been translated to the corresponding β-channels. The aim of this section is to simplify the communication structuring phase through skeletal parallel programming, and simultaneously simplify the implementation and deployment of algorithmc skeletons by using β-channels.

4.4.1 Skeletons, patterns and communication structures

Skeletal parallel programming models provide programming constructs that directly correspond to frequently occurring patterns of parallel computation, such as communication patterns in message passing parallel programs. During application development, the programmer expresses the algorithm in terms of these patterns, and representations are translated automatically into the corresponding concrete implementations through compiler transformations, or through application programming interfaces. Communication structures, on the other hand, also define the manner in which processes communicate during a computation. It is therefore

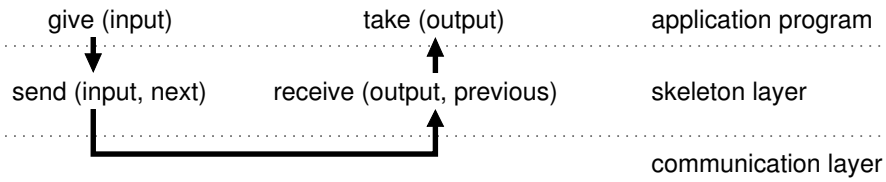


Figure 4.22: Pipeline skeleton implementation based on the introduction of a skeleton layer. This provides a higher-level abstraction where ranks of the predecessor and successor processes are resolved internally by the skeleton layer.

fair to state that skeletal parallel programming and communication structures are partially related because they both provide abstractions for the underlying communications based on the pattern of communications manifested by the algorithm that is being implemented.

4.4.2 Skeletal programming with β-channels

In Section 3.9, we described the β-channel properties which allow grouping, identification, and referencing of a group of communications. In this section, we use these properties, again, to implement algorithmic skeletons.

The simplicity of programming with algorithmic skeletons is due to the fact that complex low-level implementation of communication patterns can be concealed from the programmer by means of the abstraction provided by a *skeleton layer* (see Figure 4.22). Therefore, instead of implementing commonly occuring patterns of communications, the programmer can simply use existing skeletons that provide the necessary implementation of the pattern. In a pipeline computation, for example, two adjacent stages communicate data so that following stage uses the data received from the previous stage.

Implementing such a pipeline computation without skeletons means implementing the communications explicitly within the computation loop, which resembles the following simiplified algorithm:

```
loop
  receive(input, previous);
  output ← compute(input);
  send(output, next);
end loop
```

The values of *previous* and *next* depend on the stage-to-process mapping. For example, if stage i is mapped to process $\mathcal{P}_i$ for $1 \leqslant i \leqslant n$, then *previous* = 1 and *next* = 3 on $\mathcal{P}_2$. If we change the mapping, the values for *previous* and *next* also changes

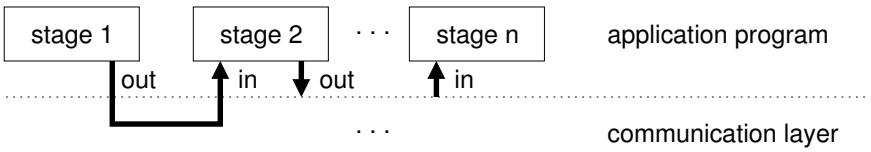


Figure 4.23: An n stage pipeline skeleton implementation which uses β-channels for communications. After creating a pipeline skeleton instance, each stage is provided with the source and sink β-channels, *in* and *out*. These β-channels can be used directly within the stage. In addition to providing the abstraction, this implementation removes the skeleton abstraction layer at runtime.

accordingly. These changes, however, are not relevant to the realisation of a pipeline computation, and therefore may be concealed from the programmer by means of a higher-level abstraction.

Higher-level abstraction is the fundamental idea behind skeletal programming. With algorithmic skeletons the above loop will be simplified as follows:

```
loop
  take(input);
  output ← compute(input);
  give(output);
end loop
```

The advantage of this algorithm is that the *previous* and *next* values are managed by the skeleton layer, which interacts with the application program when the skeleton interfaces give() and take() are invoked on each of the stages. This affects programming, and the resulting program in two ways. Firstly, the programmer need not be concerned about the stage-to-process mapping. The pipeline skeleton will handle the mapping internally without further programmer intervention, therefore simplifying the implementation. Secondly, the skeleton layer can decide the best stage-to-process mapping policy that will allow the pipeline computation to use the available resources in the most optimal manner.

In existing messaging passing skeletal programming models, algorithmic skeletons are implemented by introducing an intermediate skeleton layer between the application program and the communication layer, as shown in Figure 4.22.

Although the skeleton abstraction simplies programming, implementations that introduce an intermediate skeleton layer suffer from a degradation in performance due to abstraction overhead: for example, the skeleton interface give() internally invokes the send() communication interface after determining the value of *next*. Upon further investigation, we can observe, however, that the value of *previous* and *next* remain unchanged after a successful mapping for the entire computation loop. We

should therefore be able to remove the skeleton layer without affecting any of the advantages provided by an algorithmic skeleton. This is where the β-channel approach can be used to our advantage.

After the communication structuring phase, the communication structures manifested by the algorithm are translated into opaque data structures, the β-channels. These β-channels are used by the application during the communication activation phase, without the need to know how they were created. This means that, once the β-channels have been created, the communication structure does not affect their usage. An execution instance of the pipeline computation with β-channels is shown in Figure 4.23.

By combining skeletal programming with the β-channel approach, we can therefore achieve efficient, yet simple, higher-level abstractions. To do this, we use algorithmic skeletons to abstract the communication structuring phase, while the communication activation phase remains the same as was the case with the β-channel approach. In the following section, we demonstrate this by implementing algorithmic skeletons which uses β-channels for communication.

Implementing the pipeline algorithmic skeleton

To use skeletal programming for simplifying the communication structuring phase, we implement the following skeleton interfaces:

sk_pipe_t *__sk_pipe_create__ (bc_plist_t *pl, sk_fptr_t *fptr, sk_dmap_t *dmap);

> Creates a pipeline skeleton instance for the processes in pl, with the stage mapping in fptr, and the input and output data types for each stage given by dmap. This skeleton instance can be applied for computations on different data sets.

void **sk_pipe_exec** (sk_pipe_t *pipe, void *in, void *out);

> Executes the pipeline skeleton instance, pipe, with the input for the first stage taken from in, while the results from the last stage are stored in out.

void **sk_pipe_destroy** (sk_pipe_t *pipe);

> Destroys the pipeline skeleton instance pipe.

The code segments from the implementation of the pipeline skeleton are shown in Figure 4.24. When sk_pipe_create() is invoked, a skeleton instance is returned to the participating processes in pl. This skeleton instance is initialised with the β-channels necessary to realise the pipeline communication structure, as shown at lines 13–18 for the intermediate stage. The skeleton instance is also initialised with the corresponding stage task which the process should execute upon entering the skeleton instance, shown at line 20. When sk_pipe_exec() is invoked this task is ex-

```
1  sk_pipe_t *sk_pipe_create ( bc_plist_t *pl, sk_fptr_t *fptr, sk_dmap_t *dmap ) {
       int i := 0;
3      sk_pipe_t *pipe := NULL;
       /* If process is not in the process list 'pl', return NULL. */
5      ...
           do {
7              if ( bc_rank = pl→plist[i] ) {
                   if ( i = 0 ) { /* First stage. */
9                      ...
                   } else if ( i = pl→count − 1 ) { /* Last stage. */
11                     ...
                   } else { /* Intermediate stage. */
13                     pipe→prod := bc_plist_create ( 1, pl→plist[i−1] );
                       pipe→src := bc_src_create ( pipe→prod, dmap[i].in,
15                                                 BC_ROLE_PIPE );
                       pipe→cons := bc_plist_create ( 1, pl→plist[i+1]);
17                     pipe→sink := bc_sink_create ( pipe→cons, dmap[i].out, 1,
                                                     BC_ROLE_PIPE );
19                     pipe→role := SK_PIPE_INTER;
                       pipe→fptr := fptr[i];
21                 }
               }
23         } while ( ++i < pl→count);
           return pipe;
25     }
       void sk_pipe_exec ( sk_pipe_t *pipe, void *in, void *out ) {
27         if ( pipe→role = SK_PIPE_FIRST ) pipe→fptr ( in, NULL, pipe→sink );
           else if ( pipe→role = SK_PIPE_LAST ) pipe→fptr ( out, pipe→src, NULL );
29         else pipe→fptr ( NULL, pipe→src, pipe→sink );
       }
31     void sk_pipe_destroy ( sk_pipe_t *pipe ) {
           bc_chan_destroy ( pipe→src ); bc_chan_destroy ( pipe→sink );
33         bc_plist_destroy ( pipe→prod ); bc_plist_destroy ( pipe→cons ); free ( pipe );
       }
```

Figure 4.24: β-channel implementation of the pipeline algorithmic skeleton interface. The function sk_pipe_create() is used by application programs to create a pipeline skeleton interface. The stages in the pipeline instance are executed by invoking sk_pipe_exec(). When the pipeline skeleton instance is no longer needed, the resources are deallocated by invoking sk_pipe_destroy(). The pipeline skeleton instance contains information about the pipeline communication structure, which is represented by the source and sink β-channels.

```
1  void stage_first ( void *ivar, bc_chan_t *ibc, bc_chan_t *obc ) {
       int j := 0;
3      do { bc_var ( obc, int ) := * ( (int * ) ivar + j );
           if (bc_var ( obc, int ) ≠ 0) {
5              bc_var ( obc, int ) := bc_var ( obc, int ) + 1;
               bc_commit ( obc );
7          } else { bc_commit ( obc ); break; }
       } while ( ++j );
9  }
   void stage_inter ( void *ivar, bc_chan_t *ibc, bc_chan_t *obc ) {
11     do { bc_get ( ibc, bc_vptr ( obc, int ), 1 );
           if ( bc_var ( obc, int ) ≠ 0 ) {
13             bc_var ( obc, int ) := bc_var ( obc, int ) + 2;
               bc_commit ( obc );
15         } else { bc_commit ( obc ); break; }
       } while ( 1 );
17 }
   void stage_last ( void *ovar, bc_chan_t *ibc, bc_chan_t *obc ) {
19     int j := 0;
       do { bc_get ( ibc, ( int * ) ovar + j, 1 );
21         if ( *( ( int * )ovar + j ) ≠ 0 ) {
               *( ( int * )ovar + j ) := *( ( int * ) ovar + j ) + 3;
23         } else break;
       } while ( ++j );
25 }
   int main (int argc, char *argv[] ) {
27     sk_pipe_t *pipe;
       sk_dmap_t dmap[] := {{bc_int, bc_int}, {bc_int, bc_int}, {bc_int, bc_int}};
29     sk_fptr_t func[] := {stage_first, stage_inter, stage_last};
       bc_plist_t *plist;
31     int in[11] := {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}, out[10];
       plist := bc_plist_create ( 3, 0, 1, 2 );
33     pipe := sk_pipe_create ( plist, func, dmap );
       sk_pipe_exec ( pipe, in, out );
35     sk_pipe_destroy ( pipe );
       bc_plist_destroy ( plist );
37     return 0;
   }
```

Figure 4.25: Example usage of the pipeline skeleton interface inside an application program with a pipeline communication pattern. This application has three stages, each stage adds some value to the input data. A pipeline skeleton instance is first created by invoking sk_pipe_create(), which takes the stage functions, input and output data types for each stage, and the three processes upon which the stages will be mapped. Receiving and sending data is performed by activating the source and sink β-channels, ibc and obc. We use the interfaces, bc_var(), bc_vptr(), and bc_commit(), for sending data.

ecuted, as shown at lines 27–29. The task function for the first stage is provided as parameters the input application buffer, in, and the sink β-channel for sending data to the next stage. Similarly, the task function for the last stage is provided as parameters the output application buffer, out, and the source β-channel for receiving data from the previous stage. For the intermediate stages, only the source and sink β-channels are provided as parameters to the task function.

Using the pipeline skeleton

In this section, we discuss example usage of the above skeleton interfaces for implementing a pipeline with three stages. Here stage i executes the function f_i, defined as $f_0 : x \rightarrow x+1$, $f_1 : x \rightarrow x+2$, and $f_2 : x \rightarrow x+3$. The implementation for the three stage pipeline is shown in Figure 4.25. After the stage functions are defined at lines 1–25, the pipeline skeleton instance is created at line 33. This skeleton instance is then executed at line 34, by passing the in and out application buffers for data input and data output. The skeleton instance is subsequently destroyed at line 35.

In each stage function, the application utilises the input and output β-channels, ibc and/or obc, provided by the skeleton instance without explicitly creating them. This is possible because β-channels provide handles to communications; however, their creation can be transparently handled by the algorithmic skeleton: therefore providing a higher-level abstraction of the underlying communications.

On the other hand, by utilising the β-channels with the interfaces bc_var(), bc_vptr(), and bc_commit(), the application can directly access the underlying communication layer, bypassing the skeleton abstraction layer once the stage task begins execution. The adopted approach is different from the MPI based skeleton implementations, such as the ESKEL skeleton library [33], where stage tasks invoke skeleton interfaces which internally invoke appropriate message passing interfaces.

In the intermediate stage task (see Figure 4.25), it is interesting to note line 11 where the output buffer associated with the sink β-channel for the next stage is used for getting data from the previous stage, and is also used during the computation at line 13; before committing the values for the next stage at line 14.

Since the Gaussian elimination algorithm can be implemented with a pipeline communication pattern (see Section 4.3.1.1), we can simplify the implementation shown in Figure 4.7 using the pipeline interfaces as shown in Figure 4.26. In this implementation, we hide the details concerning creation of the β-channels. These β-channels are created by the sk_pipe_create() interfaces, which are passed to the stage function gauss_stage() when the pipeline topology instance is executed using sk_pipe_exec() (line 38). Since we are using the matrix $\mathbf{A}$ as both input and output

```
1  void gaussian_stage (void **var, bc_chan_t *src, bc_chan_t *sink) {
       int i, j, last := bc_size − 1, r := bc_rank, y := n + 1;
3      bc_plist_t *sink_pl := NULL, *src_pl := NULL;
       bc_chan_t *sink := NULL, *src := NULL;
5      /* Eliminate all the preceding equations. */
       for ( i := 0; i < r; i++ ) {
7          /* Receive preceding equations for elimination. */
           bc_get ( src, &A[i][0], y + 1 );
9          for ( j := i + 1; j < n; j++ )
               A[r][j] := A[r][j] − A[r][i] * A[i][j];
11         A[r][n] := A[r][n] − A[r][i] * A[i][y];
           A[r][i] := 0;
13         /* Pass on received equations. */
           if ( r ≠ last ) bc_put ( sink, &A[i][0], y + 1 );
15     }
       /* Preceding equations have been eliminated, divide my equation. */
17     for ( j := r + 1; j < n; j++ )
           A[r][j] := A[r][j] / A[r][r];
19     A[r][y] := A[r][n] / A[r][r];
       A[r][r] := 1;
21     /* Send my equation for elimination in the succeeding stages. */
       if ( r ≠ last ) bc_put ( sink, &A[r][0], y + 1 );
23 }
   int main(int argc, char *argv[]) {
25     sk_pipe_t *pipe;
       sk_pipe_dmap_t dmap[] := {{bc_float, bc_float}, {bc_float, bc_float},
27                               {bc_float, bc_float}, {bc_float, bc_float},
                                 {bc_float, bc_float}, {bc_float, bc_float}};
29     sk_pipe_fptr_t func[] := {gaussian_stage, gaussian_stage,
                                 gaussian_stage, gaussian_stage,
31                               gaussian_stage, gaussian_stage};
       bc_plist_t *plist;
33     /* Create process list for the pipeline with six stages. */
        plist := bc_plist_create ( 6, 0, 1, 2, 3, 4, 5 );
35     /* Create pipeline topology. */
       pipe := sk_pipe_create ( plist, func, dmap, 1 );
37     /* Execute topology. */
        sk_pipe_exec ( pipe, NULL, NULL );
39     /* Destroy the pipeline topology. */
        sk_pipe_destroy ( pipe );
41     /* Destroy the process list. */
       bc_plist_destroy ( plist );
43     return 0;
   }
```

Figure 4.26: Simplification of the Gaussian pipeline implementation Figure 4.7, using pipeline skeleton interfaces. This implementation removes the concerns related to the creation and destruction of the communication structure.

buffer, they are not passed explicitly during the invocation of sk_pipe_exec(). Within the stage functions, the β-channels provided by the pipeline instance are used immediately, as was the case with the implementation without using the pipeline skeleton interfaces. This is possible because the β-channels provide handles to the communication structures.

As we can observe in the implementation of the pipeline skeleton (see Figure 4.24), the most important part is the creation of the β-channels based on the participating process lists, supplied to the pipeline interface. Based on this simple framework, we can implement several other skeletons, by modifying the code segments where the β-channels are created. For example, to create a farm skeleton, we may create a sink β-channel with BC_ROLE_FARM role for the first process in the process list (assuming it is the farmer process), while the rest of the processes create source β-channels with the BC_PIPE_ROLE role. Similarly, for returning the calculated values to the farmer, the farmer process creates a source β-channel with BC_ROLE_HARVEST role, while the worker processes create sink β-channels with BC_ROLE_PIPE. These β-channels can then be provided to the necessary stage functions when the skeleton execution interface is invoked.

4.5 Summary

In this chapter, we have discussed the β-channel programming model. We have discussed the two phase application development process, which divides application development into *communication structuring phase* and *communication activation phase* (see Section 4.1). We then described the set of application programming interfaces for implementing message passing algorithms into executable β-channel based parallel programs (see Section 4.2); and have demonstrated their usage by implementing non-trivial parallel algorithms (see Section 4.3). Finally, we have discussed the β-channel programming model in relation to skeletal parallel programming: emphasising that the β-channel approach is advantageous for the implementation and deployment of algorithmic skeletons (see Section 4.4). We have also demonstrated how the skeleton abstraction layer may be removed at runtime by using β-channels.

CHAPTER 5

Implementation details

This chapter discusses the internal design details of the prototype runtime system. We begin by discussing the execution model in Section 5.2. Here, we describe the various functional units of the runtime system. In Section 5.3, we discuss how the links between the source and sink β-channel are established at runtime. In this section, we also explain the reasons behind the *planarity condition* (see Definition 4.2.1). In Section 5.4, we discuss the design of the high-level communication protocol used for transferring data from the sender to the receiver. Here, we introduce the asynchronous *rendezvous* communication protocol which allows automatic overlapping of computations and communications. In Section 5.5, we discuss how message buffering is integrated into the runtime system. Finally, we conclude this section, and the chapter, by presenting the algorithms executed during the interface optimisation for send-and-forget type communications.

5.1 General design decisions

The design of the runtime system constitutes a crucial step towards attaining the objective set out in Section 2.3. Two of the most fundamental design decisions are:

- The run-time system should be multi-threaded, so that automatic overlapping of computations and communications can be implemented easily. Multi-threaded processes for improving the performance of message passing interfaces are not new. Some of the previous works in this area are due to Felten and McNamee [38], and the NEXUS approach, due to Foster *et al*. [39]. What is different about our approach is that it is based on a client-server model, where threads are classified based on their functions. This allows for a process to switch between sender and receiver roles, depending on the thread which is currently active.
- Integration of message buffering into the runtime environment requires the design of a communication protocol which is defined by the message buffer characteristics. For this, we choose a *rendezvous* communication protocol, where communications are receiver initiated. The difference from existing implementations, however, is that the *rendezvous* protocol which we have implemented is asynchronous. In order to achieve this, we design the multi-threaded runtime system by keeping in mind, the design of the communication protocol. Some of the previous works on asynchronous handling of messages can be found in [70].

  Integration of buffers into the runtime systems, as integral component, while remaining programmer definable, has long been investigated in theoretical contexts. Some of these can be found in the works of Brodsky *et al*. [22]. In fact, Karp and Miller [66] have already defined a theoretical model for parallel computation based on a network of queues between the processes. They were able to derive the conditions for determinacy, termination and queueing properties of such systems.

  The following sections discuss the runtime system in detail.

5.2 Program execution and the runtime system

A β-channel application program starts execution when all of the participating processes have successfully returned from the `bc_init()` interface call. During this function call, a multithreaded runtime system is initialised on the invoking process. This runtime system has six functional units, as shown in Figure 5.1. We shall now discuss each of these functional units in detail.

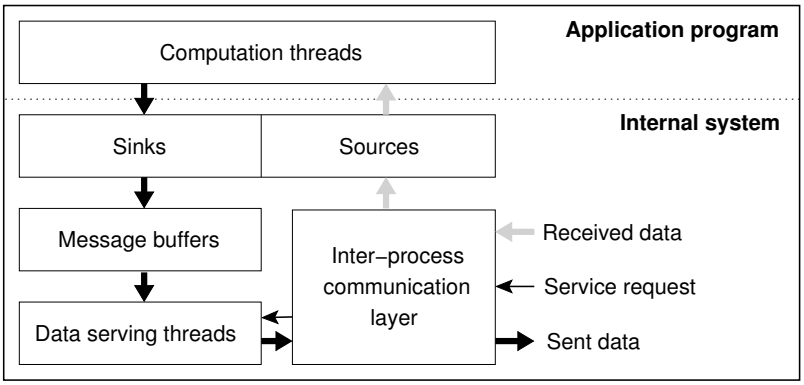


Figure 5.1: Functional units of the β-channel runtime system. The computation threads execute code segments in the application program, the data serving threads service data transfer requests from remote processes, the inter-process communication layer performs data transfers between processes, message buffers allow automatic overlapping of computations and communications, and finally, the sources and sinks contain information on the β-channels created on the host process.

Computation threads

After initialisation with bc_init(), every process is multithreaded with a set of *threads*, also referred to as *light-weight processes* [25]. These threads are categorised into two types: (1) *computation threads*, and (2) *data serving threads*. Based on the type, every thread is assigned a specific function during the execution of the application.

Execution of the code segments in the application program, with the exception of the message passing interfaces, is assigned to the computation thread. Computation threads therefore produce, or transform, data while behaving as a producer, and consume data while behaving as a consumer. These basic functions are, in essence, similar to that of a sequential process. The computations performed by a computation thread only use *in-process data*, data which exist in the local address space. When the execution reaches a point where *off-process data* is necessary for further computations, the computation thread invokes the bc_get() interface. This issues a request to the inter-process communication layer, which performs the necessary communications on behalf of the computation thread. When data is to be sent to a remote process, the computation thread invokes the bc_put(), or bc_commit() interface. These interfaces directly copy, or commit, data into the respective message buffers associated with the β-channel on which the interface was invoked. The computation thread, however, does not participate in any inter-process communications which will result in the transfer of data to the receiver processes: this is done by the data serving threads, which use the inter-process communication layer.

Data serving threads

Data serving threads, contrary to computation threads, are internal runtime system threads. They execute a predefined set of tasks in response to data transfer service requests from remote processes. This makes them independent of any application program, and therefore, transparent to the application programmer.

Through the data serving threads,[1] a computation thread is allowed to continue with further computations without participating in actual data transfers with remote processes. After initialisation, the data serving threads immediately sleep until woken up by a service request. Once a service request is received, a request for communication is issued to the inter-process communication layer, which then transfers the data from the message buffers to the requesting process. Before issuing the communication request, the data serving thread performs all the necessary sink-to-source link resolutions so that the correct buffer units are immediately available to the inter-process communication layer. When a data transfer request has been served, the active data service threads again go to sleep, until woken up by another request. What is interesting about this arrangement is that, unless required for data communications, the processor is always available to the computation threads, which can continue with computations without worrying about the inter-process communications. It is easy to observe that this provides an automatic, yet efficient, approach for overlapping computations and communications.

Inter-process communication layer

The inter-process communication layer handles all the tasks that are to do with the communication of data with remote processes. After initialisation, a K_n complete network is established between the n participating processes. This network provides the raw communication links between all of the processes. The inter-process communication layer is activated when one of the following three events happens: (1) a data transfer request is issued by the computation thread, (2) a data transfer service request is received from a remote process, and (3) a data transfer request is issued by the data serving thread. In the first case, the inter-process communication layer sends a data transfer service request to the remote process, and waits until the requested data is received from the remote process. In the second case, the service request is transferred to the respective data serving threads by waking them from their sleep. Finally, in the third case, data is transferred to the remote process by accessing the message buffers associated with the correct sink β-channel

[1] To simplify implementation of the prototype runtime system, we assign separate data serving threads to each of the remote processes with which the host process is allowed to communicate.

on the host process. The resolution of the sink-to-source link is performed by the data serving thread, prior to issuing a communication request to the inter-process communication layer. The third case, therefore, follows the second case whenever the sink-to-source link is resolved successfully.

Message buffers

Every sink β-channel created by a computation thread is associated with a message buffer. This buffer is used to store data that is produced or transformed by the computation thread. The data serving threads should send these data to the remote processes upon receiving data transfer service requests.

The *message buffers* functional unit performs all the tasks (for example, creation, destruction, etc.) related to the management of message buffers. The type of buffer associated with a sink β-channel depends on the type of the β-channel role (for example, a BC_ROLE_FARM role uses a message buffer which is shared by all the data serving threads; a BC_ROLE_SPREAD role, on the other hand, uses message buffers where separate buffer units are assigned to each of the data serving threads). This functional unit therefore performs another critical function, which is providing a uniform interface to the inter-process communication layer so that different types of buffers can be accessed easily. Furthermore, this functional unit is also responsible for ensuring mutual exclusion on the shared buffers (for example, buffers associated with BC_ROLE_FARM role).

Sinks and sources

The functional units, *sinks* and *sources*, manage creation and destruction of source and sink β-channels defined on the host process. When a computation thread invokes β-channel creation interfaces, bc_src_create() or bc_sink_create(), these functional units ensure that the information necessary to access any of these β-channels is available to all of the other functional units. In particular, the data serving threads use information stored in these functional units while resolving the sink-to-source links. From the programmer's perspective, these two functional units provide indirect access to the other two functional units: message buffers, and inter-process communication layer. In Section 3.9, we have discussed how these facilities are used to perform optimisations such as avoiding intermediate memory copy.

In the following section, it will be discussed in depth how β-channels are managed in these functional units.

5.3 Structuring communications at runtime

This section answers the question: how is the holistic communication pattern implemented concretely at runtime? We discuss how localised ordering of events is used to achieve asynchronous creation and destruction of β-channels. We also discuss, in detail, what actually happens when a program is executed. This furthers the previous discussions, by presenting an example execution instance.

Hewitt and Baker [60] axiomatised that the concept of a unique global clock is not meaningful in the context of a distributed system of autonomous parallel agents; which Clinger [31] showed was consistent with the principle of parallel processing. Following these arguments, Agha concludes [1, page 10]:

> "…for a distributed system, a *unique (linear) global time* is not definable. Instead, each computational agent has a local time which linearly orders the events as they occur at the agent, or alternatively, orders the local states of that agent. These local orderings of events are related to each other by the *activation ordering*."

Activation ordering defines the causal relationship between events happening at different agents, such that global ordering of events is a partial order in which events occurring at different computational agents are unordered unless there exists a causal relationship between the agents.

In relation to the structuring of communications at runtime, the above observation is important on two counts. Firstly, it says that a unique global state is not definable. Secondly, unless there exists a causal relationship between any two processes, both processes can execute asynchronously without invalidating the partial ordering of global events. If we recall our discussion on the abstraction of a holistic communication pattern (see Chapter 3), we can observe that the communication pattern expressed by a communication structure represents a global state. However, based on Agha's conclusion, this cannot be defined as a 'unique' global state. The global state should therefore be broken down into localised components which are ordered based on the ordering of local events—leaving the partial global ordering to the dependency edges that are created by a sink-to-source link.

From the above discussions, it is clear that our approach of abstracting holistic communication patterns in terms of process specific localised communication patterns is theoretically sound. Furthermore, management of the β-channel is already asynchronous because none of the interfaces introduced in Section 4.2 depended on the assumption of inter-process synchronisation. If synchronisation is necessary (for example, waiting for data to be transferred by the sender), it is handled internally

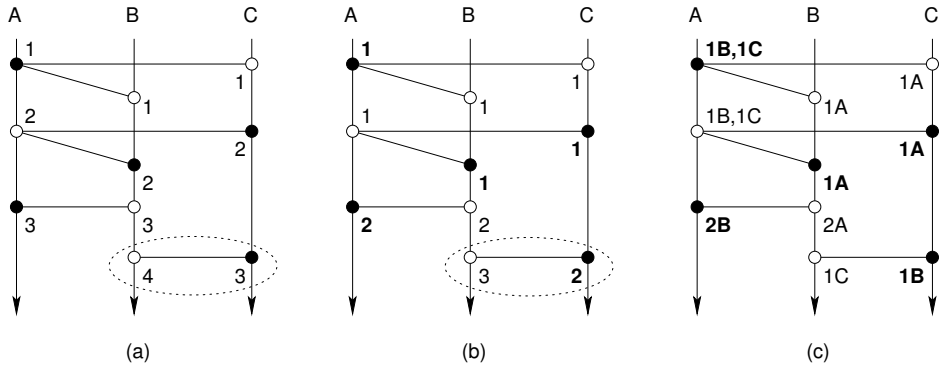


Figure 5.2: Tag assignment policy based on activation ordering: (a) without incorporating β-channel type and remote process information, (b) without incorporating remote process information, and (c) both β-channel type and remote process information is used. • and ○ respectively represent *sink* and *source* β-channels. In (b) and (c), bold numerals represent tags assigned to sink β-channels, while numerals in normal text represent tags assigned to source β-channels.

by the runtime system. Of course, such synchronisations only occur when there is a causal relationship between two processes, and therefore the global partial ordering is maintained throughout the execution of the program.

5.3.1 Establishing the sink-to-source link

Creation and destruction of β-channels on different processes can happen asynchronously, even when the β-channels belong in the same communication structure. When a process does not create more than one β-channel during the entire execution, implementing the communication structure is quite straightforward as the sink-to-source links can be established directly by assigning the same identification tag over all of the β-channels. This situation, however, becomes complicated when a process creates more than one β-channel. The question is: how do we assign identification tags that will provide a consistent tagging policy, so that any sink-to-source link can be resolved correctly at runtime?

One naive approach is assigning identification tags based on the local ordering of events which corresponds to the creation of a β-channel, as shown in Figure 5.2.a. Let us ignore for the moment the difference between the source and sink β-channels. The first two β-channels created on each of the three processes are consistently tagged: both β-channels are respectively assigned tags 'one' and 'two'. When we reach the stage where $\mathcal{P}_A$ and $\mathcal{P}_C$ create the third β-channel, $\mathcal{P}_B$ has created two β-channels. The tags across $\mathcal{P}_A$ and $\mathcal{P}_B$ are still consistent because they have the same tag, 'three'; with this, the sink-to-source link can still be resolved.

The tags across $\mathcal{P}_B$ and $\mathcal{P}_C$ are, however, not consistent: the fourth β-channel on $\mathcal{P}_B$ has identification tag 'four', while the third β-channel on $\mathcal{P}_C$ has identification tag 'three'. If we try to resolve the sink-to-source link between $\mathcal{P}_B$ and $\mathcal{P}_C$ at run-time, we are faced with an error due to these inconsistent tags. The conclusion, therefore, is that the tagging policy should account for the variation in the number of β-channels created by each of the processes.

Based on the conditions for the validity of a communication structure (see Definition 3.7.5), we know that for every sink β-channel, there is always a corresponding source β-channel, and *vice versa*. We can use this condition to our advantage for refining the tagging policy just discussed. Instead of treating all the β-channels equally while assigning tags, the new tagging policy separates the tags assigned to source and sink β-channels. This means that source β-channels are assigned tags from a set of identification tags, independent of the set of tags assigned to the sink β-channels. According to this tagging policy, two β-channels on the same process can therefore have the same identification tags as long as they both represent different β-channel types.

The refined tagging based on the new policy is shown in Figure 5.2.b. The **bold** numerals represent tags assigned to sink β-channels, while numerals in normal text represents tags assigned to source β-channels. Although the tagging policy is more sophisticated, it still does not solve the problem. The problem arises because the number of source and sink β-channels created on a process does not differentiate between the remote processes with which the sink-to-source link should be established. We can see this in Figure 5.2.b, where the final two source β-channels created on $\mathcal{P}_B$ are assigned consecutive tags, without acknowledging the tags assignment to the β-channels on the remote processes, $\mathcal{P}_A$ and $\mathcal{P}_C$.

We now differentiate between source and sink β-channels by incorporating remote process information. With this tagging policy, source and sink β-channels are assigned independent tags, and within the set of β-channels of a given type, the tags are assigned based on the remote process with which the sink-to-source link is established. This ensures that β-channel tags are also grouped according to the remote process. In Figure 5.2.c, we show tag assignment based on this policy. Each tag now contains the remote process information. If we consider, for example, the first β-channel created on each of the three processes, the β-channel of $\mathcal{P}_A$ is assigned the tag '**1B,1C**'. This tag means that the sink β-channel links to two source β-channels on the remote processes, $\mathcal{P}_B$ and $\mathcal{P}_C$, with source tag values of 'one'. On the processes, $\mathcal{P}_B$ and $\mathcal{P}_C$, the tag value '1A' means that the source β-channel is linked to a sink β-channel on $\mathcal{P}_A$ which has a sink tag value of 'one'. As we can

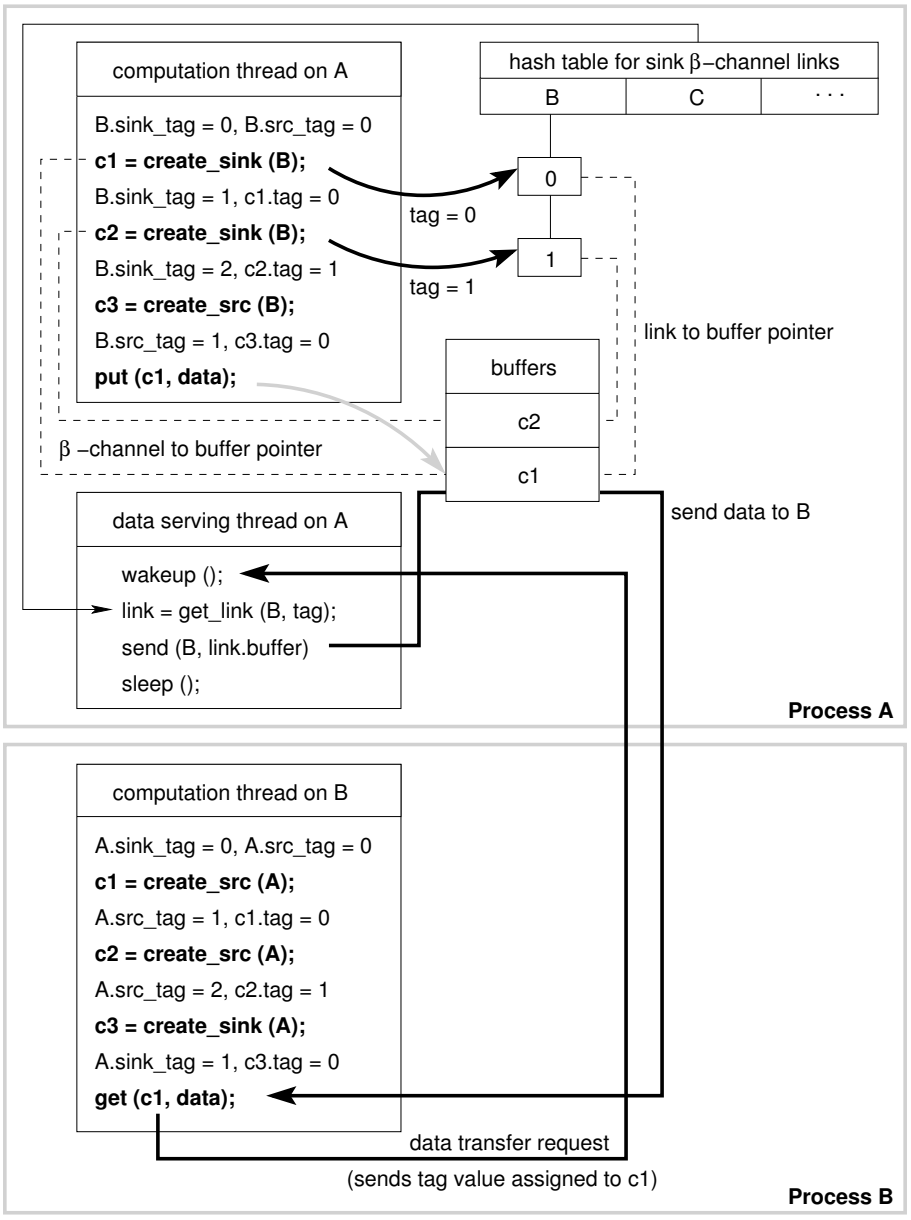


Figure 5.3: Example execution of a communication structure based application program. This shows what happens in the different functional units of the runtime system when an application program is executed. We show the creation of source and sink β-channels, invocation of bc_put() and bc_get(); and the events that lead to the transfer of data between the two processes.

see, this solves the tagging problem. This tagging policy is easily implemented with a *hash table*.

5.3.2 Example execution of an application program

In Figure 5.3, we show the execution of a β-channel application program with two processes, $\mathcal{P}_A$ and $\mathcal{P}_B$. $\mathcal{P}_A$ creates two sink β-channels, c1 and c2, and a source β-channel, c3. $\mathcal{P}_B$, on the other hand, creates two source β-channels, c1 and c2, and a sink β-channel, c3. The code segments executed by the computation threads are shown within the respective functions unit. The hash table only stores information on the sink β-channels because this is the only information necessary for finding out which message buffer should be accessed while transferring data to a remote process. All the information relevant to the source β-channel can be stored in itself.

During the communication structuring phase, when the sink β-channel, c1, is created, a link is inserted into the hash table. This link is assigned a tag value which equals sink_tag. Because different sink_tag values are maintained separately for each of the remote processes, this link is assigned the value of B.sink_tag, as it corresponds to the remote process $\mathcal{P}_B$. A message buffer is then allocated for this sink β-channel, and appropriate pointers are updated within c1 and the hash table link so that the message buffer can be accessed by the computation thread (while putting data), and the data serving thread (while serving data transfer requests). These pointers are represented by dashed lines. After successfully creating a sink β-channel, the value of B.sink_tag is incremented. We repeat the same process for the second β-channel, c2. When the source β-channel, c3, is created, instead of using B.sink_tag, B.src_tag is used. In addition, no link is inserted into the hash table because source β-channels are only used by the computation thread for making a request to the inter-process communication layer. All the relevant information can therefore be stored in the source β-channel. Similar analysis can be done on $\mathcal{P}_B$.

During the communication activation phase, $\mathcal{P}_A$ puts data from data into the sink β-channel, c1, by invoking bc_put(). The message is then received on $\mathcal{P}_B$ by invoking bc_get() on the source β-channel, c1. When putting data into the sink β-channel, the values are transferred directly to the message buffers pointed to by c1. This is shown by the grey arrow. When the bc_get() interface is invoked on $\mathcal{P}_B$, a request is sent to $\mathcal{P}_A$. This wakes up the data serving thread on $\mathcal{P}_A$. The link is then resolved by using the hash table information, as shown by the function get_link(). This link is passed to the inter-process communication layer, which is used to access the message buffer pointer while transferring the data to $\mathcal{P}_B$. The data serving thread then goes to sleep.

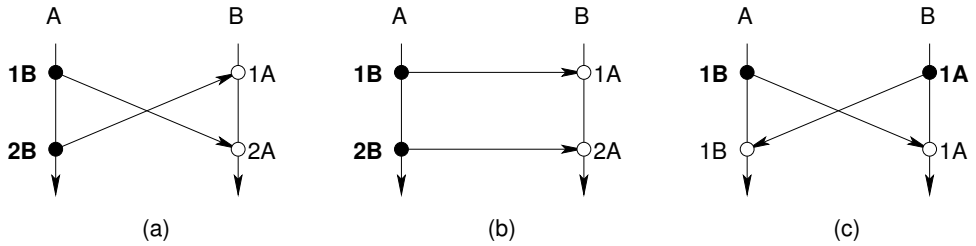


Figure 5.4: The reason for the planarity condition (see Definition 4.2.1). (a) When the dependency edges with the same direction cross each other, we observe a mismatch of the source and sink β-channel tags, (b) the mismatch is resolved by removing the cross over, and (c) crossing over of dependency edges with different directions of data flow does not pose a problem because the resulting tag values are not affected by it. Normal labels represent source tags; bold labels sink tags.

5.3.3 Why do we need 'the planarity condition'?

The *planarity condition* states that no two dependency edges between two partitions, with the same direction of data flow, should cross each other when represented with 'straight lines' (see Definition 4.2.1). In this section, we discuss why this condition is necessary for the prototype implementation of the runtime system.

In Section 5.3.1, we have discussed the tag assignment policy used to establish sink-to-source links at runtime. This tagging policy uses the ordering of local β-channel creation events to determine the tag values. Due to this approach, the resolution of sink-to-source links becomes problematic when the planarity condition is defined. To clarify this, let us consider a case where the dependency edges cross each other, as shown in Figure 5.4. In the first figure, case (a), two processes, $\mathcal{P}_A$ and $\mathcal{P}_B$, create two separate communication structures, each represented by a β-channel. $\mathcal{P}_B$ creates the source β-channels in reverse order, so that the dependency edges cross. If we see the resulting tag values, we can observe that the sink and source β-channel tag values are mismatched. For example, the tag value for the first sink β-channel does not match the tag value assigned to the corresponding source β-channel ('1B' does not match '2A'). In a case where the planarity condition is satisfied, as shown in case (b), we do not observe this mismatch. The source and sink tags in both communication structures match each other. Finally, in case (c), when the direction of data flow is opposite, we can ignore the planarity condition. As we can see, the tags for the source and sink β-channels match, even when the dependency edges cross each other.

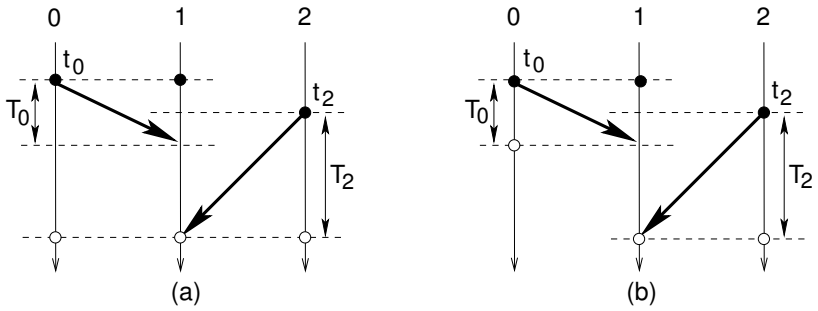


Figure 5.5: Data collection on $\mathcal{P}_1$ from $\mathcal{P}_0$ and $\mathcal{P}_2$. (a) strictly eager protocol (no buffering) (b) lenient eager protocol (buffering on receiver). • represents interface invocation, and ○ represents return.

5.4 Communication protocol

In message passing systems, the communication protocol[2] used by the communication interfaces determines the progression rule: how should a process continue after invoking a message passing interface? It is therefore important to define the communication protocol used by the message passing interfaces. In existing systems, the interfaces are classified into synchronous and asynchronous interfaces based on the communication protocol used to implement them.

5.4.1 Synchronous interfaces

With *synchronous* interfaces, once a process initiates a communication it cannot continue until that communication has completed successfully. Such interfaces use the *eager* communication protocol, and are often referred to as *blocking* interfaces. What is considered to be 'completion' of the communication is usually defined by the semantics of the underlying implementation; a detailed discussion of which can be found in Cypher and Leu's paper [35].

Figure 5.5 shows two ways of implementing a collective communication between three processes, $\mathcal{P}_0$, $\mathcal{P}_1$, and $\mathcal{P}_2$, where $\mathcal{P}_1$ collects data from $\mathcal{P}_0$ and $\mathcal{P}_2$. Both implementations use different versions of the *eager* point-to-point communication protocol [53].

In Figure 5.5.a, the implementation is based on a strictly *eager* communication protocol, where the function returns only when the entire collective communication has been completed successfully. Assuming that $\mathcal{P}_0$ and $\mathcal{P}_2$ initiated *send* at times t_0 and t_2 respectively, where $t_0 < t_2$; and that it takes T_0 and T_2 time units for $\mathcal{P}_1$

[2] In this dissertation, the term 'communication protocol' will only refer to the protocol employed by the highest level communication layer, which may be implemented over a lower level protocol layer.

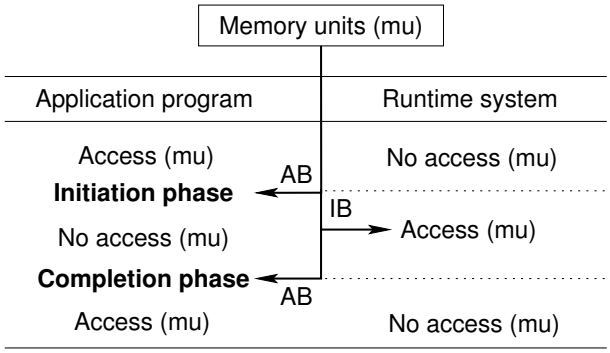


Figure 5.6: Switching memory unit abstraction in split-phase communication protocol.

to complete receiving data from $\mathcal{P}_0$ and $\mathcal{P}_2$ respectively, then $\mathcal{P}_0$ has to wait for $T_2 + (t_2 - t_0) > T_0$, with a wastage of $T_2 + (t_2 - t_0) - T_0$ time units. We improve this situation, as shown in Figure 5.5.b, by utilising a lenient eager protocol that buffers data at $\mathcal{P}_1$, allowing $\mathcal{P}_0$ to continue once the data is received by $\mathcal{P}_1$. Similar buffering is done for the data received from $\mathcal{P}_2$. In this manner, the wastage on either of the sending processes can be minimised by buffering data on the receiving process.

In both versions of the eager protocol, however, the time units T_0 and T_2 required to complete the individual communications vary depending on when $\mathcal{P}_1$ initiated the *receive* interface: a delay by $\mathcal{P}_1$ affects the waiting time on the sending processes because they cannot continue until the data is accepted by $\mathcal{P}_1$. This means that the producers and the consumers are tightly coupled. Hence, assuming that $\mathcal{P}_0$ and $\mathcal{P}_2$ initiated *send* interfaces at time t_0 and t_2, with $t_0 < t_2$, and $\mathcal{P}_1$ initiated *receive* at time $t_1 > t_0, t_2$, the wastage suffered by $\mathcal{P}_0$ and $\mathcal{P}_2$ in Figure 5.5.b is, respectively, $t_1 - t_0$ and $t_1 - t_2$. Within a loop, this wastage is multiplied by the number of iterations, which can eventually become a serious performance bottleneck [49].

5.4.2 Asynchronous interfaces

With *asynchronous* interfaces, processes are allowed to continue computation by deferring communications until the remote processes are ready; or, by allowing communications to proceed simultaneously with the computations. These interfaces are often referred to as *non-blocking* interfaces because the function returns immediately without waiting for the completion of the communication. The following two conditions are, however, imposed: (1) the invocation of the non-blocking interfaces should be followed by a corresponding test for completion, and (2) the associated application buffer should not be utilised before the test condition is satisfied.

Cypher and Leu [35] describe the semantics of such communications in terms of

split-phase communication protocols, where a communication is divided into two phases: initiation phase, and completion phase. During the initiation phase, the runtime system is issued with a communication request which it should perform on behalf of the calling process. The calling process resumes further computation without waiting for the communication. When the application buffer associated with the previous communication is required by the calling process, it tests for the completion of the communication before re-utilising the buffer during computations.

By introducing the initiation and completion phases, the split-phase communication protocol uses a subtle form of message buffering (see Figure 5.6). This involves switching the abstraction of the memory units from an application buffer (AB), which is accessible only to the application program, to an implementation buffer (IB), which is accessible only to the runtime system, and back.

5.4.3 Asynchronous *rendezvous*

In the β-channel programming model, all the communication interfaces are considered to be asynchronous: the level of asynchrony is determined by the buffer size. By explicitly integrating message buffering within the programming model, both synchronous and asynchronous interfaces are unified, and use the asynchronous *rendezvous* communication protocol defined as follows:

Definition 5.4.1 (**Asynchronous rendezvous protocol**)
In the *asynchronous rendezvous protocol*, two processes communicate data based on a request-service scheme—apparent in client-server models—where the sender (server) sends data only when the receiver (client) makes a request. The additional condition, however, is that this should happen asynchronously so that neither the sender nor the receiver waits for the other when the data to be sent is already available in the message buffer. ▌

Implementing the request-service scheme [71, page 125] does not pose a significant problem as it has already been implemented as programming language constructs in ADA [15]; the real concern is providing asynchrony. The rendezvous is synchronous in ADA—either sender or receiver has to wait for the other before continuation. As we have seen previously (see Section 5.4.1), making a sender wait for the receiver is inefficient. Forcing the receiver to wait for the sender is also inefficient: consider a case where the sender has continued with further computation because the receiver did not make a request for the available data, following which the receiver makes a request and finds the sender busy with computation. The main

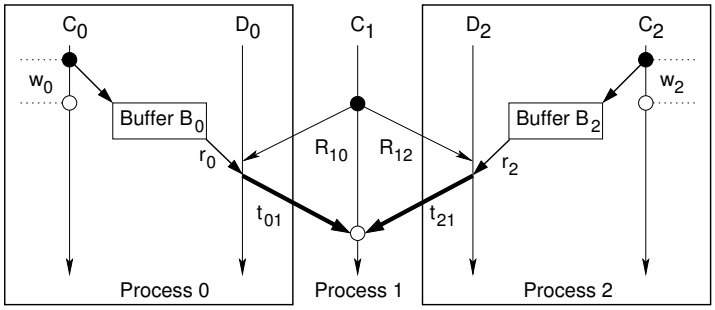


Figure 5.7: Asynchronous *rendezvous* protocol: data collection on $\mathcal{P}_1$ from $\mathcal{P}_0$ and $\mathcal{P}_2$. C_i and D_i respectively represent the computation thread and data serving thread on $\mathcal{P}_i$. • represents interface invocation, and ○ represents return.

challenge therefore is to decouple the sender and the receiver.

The asynchronous rendezvous protocol as implemented in the β-channel runtime system is shown in Figure 5.7. The sink β-channels implement buffers, B_0 and B_2, on the senders, where data are stored until they are requested by the receiver. As the buffering is done on the sender, a bc_put() on the corresponding β-channel is equivalent to a local memory access. Therefore, if the buffer was not full at the time of the invocation, the latency of bc_put() should equal cw_i where w_i is buffer write time and c is some constant. This decouples the sender from the receiver, leaving the buffer as an indirect dependency, the strength of which is determined by the buffer size. On the other hand, when a request for data is received from the consumer, the transfer of data is handled by the data serving threads (see Section 5.2). As the data serving threads on the producer are always waiting for data transfer requests, every request for data is served immediately as long as the buffer is not empty. This decouples the receiver from the producer.

Although the asynchronous rendezvous protocol efficiently decouples the sender and the receiver, it increases the number of communications necessary for receiving data from the sender. Each receive interface invocation consists of sending a data transfer request, and then receiving the requested data. Therefore, the bc_get() latency is given by $R_{10} + R_{12} + r_0 + r_2 + t_{01} + t_{21}$, where R_{ij} is the data request transfer time from C_i to D_j, r_i is the buffer access time on D_i, and t_{ij} is the data transfer time from D_i to C_j. If data are already available in the sender buffer, r_i approaches memory read time. t_{ij} is always incurred in any case because of the actual data transfer, hence this does not arise due to the protocol adopted. The main performance bottleneck, therefore, is the extra communication cost R_{ij}. That said, it can however be argued that R_{ij} can be reduced by exploiting the following properties:

1. Data service request messages are of constant length.

2. They are smaller compared to the actual data being requested. From the graph shown in Figure 6.10, the time to service data transfer requests becomes negligible when the size of the requested data becomes large—often the case with practical applications.
3. They can be given higher priority like *Out-of-band* data [43][94].

In the next section, we will discuss the integration of message buffers into the runtime system.

5.5 Integrating message buffers within the runtime system

In the previous section, we saw that message buffering plays an important role in determining the progression rule of the message passing interfaces. In this section, we will elaborate more on message buffering, and describe the internal details of the interface optimisations for send-and-forget type communications.

Although message buffering constitutes a significant factor in deciding the runtime behaviour of message passing interfaces, their implementation in popular message passing environments have the following limitations:

1. In some systems message buffering is considered to be entirely the programmer's responsibility. This provides the programmer with some flexibility in devising the best approach, however, they are more prone to programming errors. Managing the message buffer—creation, maintenance, and deallocation—adds additional programming concern which could have been abstracted by the runtime system. Additionally, because the message buffering is separated from the runtime system, programming an efficient message buffering system which interacts efficiently with the runtime system can be a challenging task.
2. In other systems, such as LINDA [45], message buffering is integrated, but concealed from the programmer; thus providing a high-level abstraction of the message buffers that is less prone to programming errors. This, however, prevents the programmer from making certain optimisations based on the algorithm being implemented.

   One such optimisation is based on *selective buffering*. In parallel applications, the overall frequency of the communications are not uniformly distributed. Some communications are more frequent than others; therefore, it makes sense to utilise more buffering space for the frequent ones. For example, if an application requires a single point-to-point communication between $\mathcal{P}_A$ and $\mathcal{P}_B$, while

it also requires n point-to-point communications between $\mathcal{P}_A$ and $\mathcal{P}_C$ (say communication within a for loop), it is sensible to provide more buffering for the n communications between $\mathcal{P}_A$ and $\mathcal{P}_C$. Such optimisations require interfaces that support selective buffering, allowing dynamic allocation and deallocation of message buffers for specific sets of communications.

3. In systems such as the MPI [93], message buffering is more sophisticated than the ones discussed previously. MPI supports buffering by providing two forms of buffering: (1) *standard mode* interfaces that use message buffers implemented by the runtime system, and (2) *buffered mode* interfaces that use message buffers that are provided by the programmer. Although this provides a certain level of improvement, it has the following drawbacks.

   MPI does not guarantee buffering. It is therefore the programmer's responsibility to ensure enough buffering space is available to the runtime system before invoking a communication interface. If standard mode interfaces are used there is therefore a possibility for failure if the data units being transferred are larger than the buffer space allocated by the MPI implementation. This leads to another programming concern because the MPI standard does not specify a minimum buffer size that should be supported; the size of the standard buffers therefore varies across implementations. Alternatively, by using buffered mode interfaces, a programmer can ensure that enough buffering space is available to the runtime system. However, this is again limited by the lack of selective buffering: a programmer can attach only one buffer to the runtime system which is used for all of the communications. The buffer management is therefore generic, and may not be an attractive option for certain applications for which specialised buffering is possible.

The β-channel runtime system functional unit *message buffers* resolve some of these issues as follows:

1. As sink β-channels are created and destroyed at runtime, their associated buffers are also created and destroyed dynamically. Dynamic buffering is implemented by default, without further programmer intervention. All that is needed from the programmer is the size of the buffer that is required for a set of communications which use the specified β-channel. While discussing the practical advantages of the β-channel approach (see Section 3.9), we have already shown how selective buffering is supported by the β-channel approach.
2. Embedding the message buffer within the β-channel makes programming more structured and less error prone, as the allocation, maintenance and deallocation

of the associated resources are handled automatically in accordance with the actions applied to the β-channel.

3. Resources are allocated only when it is necessary. Where message buffering is handled transparently, as in LINDA or standard mode MPI interfaces, the size of the message buffer is defined by the runtime, independent of the application being executed. This means that the resources are always reserved: whether they are needed or not. We can consider this as an inefficient usage of the available resources. Buffers allocated within β-channels, on the other hand, are reserved during the life-time of the corresponding β-channel, and hence it can be optimised for efficient resource usage by claiming only the resources that are absolutely necessary; and freeing them when not needed (for example, by invoking the interfaces bc_plist_destroy(), bc_chan_destroy() etc.).

5.5.1 Optimisation for send-and-forget communications

In Section 3.9.1 and Section 4.2, we discussed an interface optimisation for send-and-forget type communications where sent data are not re-used by the sender. In this section we provide the implementation details.

A performance concern related to message buffering is the intermediate memory copy involved in transferring data from the application buffer to the runtime implementation buffer. Copying a message of n unit size takes $O(n)$ time units, and therefore, this degradation can become a serious performance bottleneck. We argued, however, that this overhead may be considered negligible in situations where buffering the messages increases the asynchrony of the communicating processes. In fact, in some of the MPI implementations, asynchronous communication interfaces buffer smaller messages (size $\leqslant 2^{16}$), as discussed in Section 6.2.1.

The following algorithms allow applications to avoid intermediate memory copy by directly accessing the buffer units within the message buffers, as if they were application buffers. From the programmer's perspective, this is done by using the interfaces bc_var(), bc_vptr(), and bc_commit(). The first algorithm, alg_commit(), is executed within the runtime system, internally, when the computation threads invoke bc_commit() on a sink β-channel. The second algorithm, alg_send(), is executed by the data serving thread when a data transfer service request is received from a remote process. To prevent out of bound buffer access errors, only one data unit can be committed or retrieved from the buffer during each interface invocation.

```
1  void alg_commit ( queue_t *q, int count ) {
       pthread_mutex_lock ( &( q→lock ) );
3      q→rc[q→ptr.idx] := count; /* Reset reference count. */
       /* Update buffer unit pointer. */
5      q→ptr.var += q→size;
       if ( q→ptr.var = q→end ) q→ptr.var := q→start;
7      /* Check if it is safe to return. */
       i := ( q→ptr.idx + 1 ) % q→qsize;
9      while ( 1 ) {
           if ( q→rc[i] = 0 ) { /* Valid buffer unit found. */
11             q→ptr.idx := i;
               pthread_cond_broadcast ( &( q→cond ) ); /* Alert new data. */
13             pthread_mutex_unlock ( &( q→lock ) );
               return;
15         }
           pthread_cond_broadcast ( &( q→cond ) ); /* Alert new data. */
17         pthread_cond_wait ( &( q→cond ), & (q→lock ) );
       }
19 }
```

Figure 5.8: Algorithm for committing data to the buffer. ptr.idx gives the index of the buffer unit which is being committed. Index values lie within the range $[0, b)$, where b is the total number of buffer units existing within the buffer.

```
1  void alg_send ( queue_t *q, vptr_t *cptr ) {
       pthread_mutex_lock ( &( q→lock ) );
3      /* Check data availability. */
       while ( 1 ) {
5          if ( q→rc[cptr→idx] > 0 ) ∧ ( q→ptr.idx ≠ cptr→idx )
               break;
7          pthread_cond_wait ( &( q→cond ), &( q→lock ) );
       }
9      /* Send data from buffer unit. */
       send_data ( cptr→var, q→size );
11     q→rc[cptr→idx]−−; /* Decrement reference count. */
       /* Update buffer unit pointer. */
13     cptr→var += q→size;
       if ( cptr→var = q→end ) cptr→var := q→start;
15     cptr→idx := ( cptr→idx + 1 ) % q→qsize;
       pthread_cond_broadcast ( &( q→cond ) ); /* Alert free buffer unit. */
17     pthread_mutex_unlock ( &( q→lock ) );
   }
```

Figure 5.9: Algorithm for sending data from the buffer. This algorithm is for message buffers where a data unit committed into the buffer is replicated on all of the remote processes. While checking data availability, we also check if the buffer unit pointer maintained separately on the data serving threads overtakes the one maintained within the queue (the predicate q→ptr.idx ≠ cptr→idx).

Algorithm for committing data to the buffer

The complete algorithm for committing data to the buffer is shown in Figure 5.8. Two parameters are passed to this algorithm: (1) queue, which gives the message buffer associated with the sink β-channel, and (2) count, the number of data serving threads which depends on queue. After locking access to queue, the reference count, which keeps track of the number of references that have been made on the current buffer unit, is reset to count. This commits the data to the buffer. By resetting the reference counter to count, we mark availability of a new data unit within the buffer. The buffer unit pointer iptr.var, which is what bc_var(), or bc_vptr(), expands to is updated to the next available buffer unit. If there are data serving threads waiting for data, they are alerted (lines 12–13, and 16–17). Before returning, the algorithm waits until iptr.var is actually pointing to an empty buffer unit (the while loop). Due to this, the algorithm requires the message buffer queue to have at least two buffer units so that bc_var() and bc_vptr() always points to an empty buffer unit after a successful commit.

Algorithm for transferring data from the buffer

Upon receiving a data transfer service request from a remote process, the data serving thread executes alg_send(), shown in Figure 5.9. Two parameters are passed to this algorithm: (1) queue, which gives the message buffer associated with the link in the hash table (see Section 5.3.2), and (2) cptr, the data serving thread specific pointer, which marks the buffer unit from which the next request for data should be served. After locking the access to queue, the algorithm checks if data units are available in the buffer unit pointed to by cptr (the while loop). If data units are not available, the algorithm waits on the conditional variable cond. If data units are available, or when new data units are committed, the while loop breaks. Data from the buffer unit are then sent to the requesting process. To mark consumption, the reference counter for that buffer unit is decremented by one. The pointer within the message buffer, cptr, is updated so that it points to the next buffer unit. If there are other threads waiting on the conditional variable, cond, they are woken up (lines 16–17) before the algorithm returns.

The algorithm discussed in the previous two sections is used to implement shared buffers, where a data unit is used by many remote processes (the reason for updating the value of the reference counter to count for every new data unit that has been committed). These types of buffers are used in implementing roles such as

BC_ROLE_REPLICATE. By making slight modifications to the structure of queue, and the above algorithms, we can achieve different types of queue properties. For example, if the current buffer unit pointer, cptr, which is maintained separately on each of the data serving threads, is maintained within the queue; and the reference counter is set to 'one' for every new buffer units that is committed, we get the buffer queue required by roles such as BC_ROLE_FARM, where the same data unit is not used by more than one remote process.

5.6 Summary

In this chapter, we have discussed the implementation details of the β-channel runtime system. We began by describing the functional units of the β-channel runtime system (see Section 5.2), and discussed their functions in the execution of a β-channel application program. We then discussed how the holistic communication pattern represented by a communication structure is realised concretely at runtime (see Section 5.3). We discussed how the sink-to-source links between any two processes are resolved at runtime (see Section 5.3.1). We then illustrated the interactions between the functional units by using an example application program (see Section 5.3.2), following which justification for the *planarity condition* was discussed (see Section 5.3.3). In Section 5.4, we discussed the communication protocols used by existing message passing interfaces, and contrasted their qualities to those of the asynchronous *rendezvous* communication protocol (see Section 5.4.3). The remainder of this chapter focused on the integration of message buffering into the runtime system (see Section 5.5). Finally, we presented the algorithms executed by the interface optimisations for send-and-forget type communications (see Section 5.5.1).

CHAPTER 6

Evaluation

IN THIS CHAPTER, we evaluate the communication structure approach. We compare the β-channel abstraction and programming model with that of a popular message passing system. The Message Passing Interface (MPI) is a standardised message passing system, and since it is also arguably the most popular message passing system currently available, it is used as our reference system.

The evaluation is divided into two parts. The first part in concerned with the qualitative properties of the β-channel approach. This highlights the following properties: non-ambiguity, expressiveness, uniformity, and extensibility (see Section 1.2). The second part evaluates the quantitative properties. This highlights the performance of the β-channel runtime system as compared to the standard MPI interfaces.

The performance evaluations are further divided into two sections: (1) *micro-benchmarking*, where we evaluate the performances of individual interfaces; and (2) *macro-benchmarking*, where we evaluate the overall performance of an application which uses a set of interfaces.

The evaluation results show that the β-channel approach offers significant advantages over the MPI approach in terms of programmability (see page 143). With regards to performance, empirical results show that the β-channel interfaces perform better than the MPI interfaces when there is reduced contention, and are at least comparable to them during contention (see Section 6.2). The macro-benchmarking result (see Section 6.2.3) shows that implementation of an algorithm with a combination of β-channel interfaces surpasses some of the MPI implementations.

6.1 Qualitative evaluation

In this section, we evaluate the β-channel approach qualitatively, and discuss how the β-channel programming interfaces are more flexible and programmable than the MPI interfaces. While making comparisons, we focus on the following qualities: non-ambiguity, expressiveness, uniformity, and extensibility.

In Section 1.1, we introduced a simple synthetic example (see Example 1.1.1) to clarify the discussions. However, in order to make the comparisons and arguments more persuasive, we choose a real application which shows communication patterns with overlapping communication domains. This application is the mean value analysis of queueing networks.

Mean value analysis

The mean value analysis algorithm [89] is used to solve the queue length, throughput, response time etc. of a multi-class *closed*[1] queuing network [16]. This algorithm offers improvements on the first efficient approach based on the 'convolution algorithm' due to Buzen [27]. From a parallel programming perspective, what is interesting about the mean value analysis algorithm is that the problem space expands rapidly when the number of classes and their populations are increased to a reasonably large value. In general, to compute the residence time values for a load intensity vector $\vec{N} = (N_1,\ldots,N_r,\ldots,N_R)$, for a model with R classes with class r population N_r, we need to compute the queue lengths for the load intensity vectors $\vec{N}-\vec{1}_1,\ldots,\vec{N}-\vec{1}_r,\ldots,\vec{N}-\vec{1}_R$, where $\vec{1}_r$ represents a vector where all the components are zero, except for the r^{th} component, which is one. In order to provide a reasonable computation time, approximation algorithms based on the mean value analysis algorithm have therefore been suggested [91, 23].

Our aim here is to parallelise the mean value analysis algorithm, and show by implementation why the β-channel approach is more programmable as compared to the standard MPI interfaces. Gennaro and King have previously suggested parallelisation of the mean value analysis algorithm [46] by decomposing the problem space. In their approach, the initial problem space is decomposed into sub-spaces so that each sub-space can be assigned to a separate process. The decomposition is done on the reference class (normally the first class in the queueing network), and each sub-space is assigned to a separate process, where all the processes are aligned to form a *pipeline*. Even though this parallel implementation improves the

[1] Queueing network models with a fixed number of requests per class are often referred to as a *closed models*. It is usually marked by a 'feedback loop' where no requests leaves or enters the model.

performance of the sequential implementation, when we consider large number of classes with sizable populations, it is still hindered by a large problem space. Furthermore, if the population of the reference class is small, the decomposition results in more inter-process communications, which causes the communication overhead to surpass the improvement due to parallelisation.

To improve the above situation, we parallelise Schweitzer's approximation algorithm [91]. What is interesting about this algorithm is that it allows control over the decomposition of the problem space, depending on the number of processes available for the computation, so that each class can be assigned to a process—contrary to decomposition of the problem space based on a reference class.

A simplified representation of the Schweitzer's sequential approximation algorithm, adapted from [77, page 361], is given below:

$\vec{N} \leftarrow (N_1, \ldots, N_r, \ldots, N_R)$;
for $r \leftarrow 1$ to R **do**
  for $q \leftarrow 1$ to Q **do**
    $n^e_{q,r}(\vec{N}) \leftarrow N_r / K_r$;
  end for
end for
repeat
  for $r \leftarrow 1$ to R **do**
    for $q \leftarrow 1$ to Q **do**
      $n_{q,r}(\vec{N}) \leftarrow n^e_{q,r}(\vec{N})$;
    end for
  end for
  for $r \leftarrow 1$ to R **do**
    for $q \leftarrow 1$ to Q **do**
      $n_i(\vec{N} - \vec{1}_r) \leftarrow [N_r - 1] n_{q,r}(\vec{N}) / N_r + \boxed{\sum_{t=1 \wedge t \neq r}^{R} n_{q,t}(\vec{N})}$;
      if *delay queue* **then**
        $R'_{q,r}(\vec{N}) \leftarrow D_{q,r}$;
      else
        $R'_{q,r}(\vec{N}) \leftarrow D_{q,r}[1 + n_q(\vec{N} - \vec{1}_r)]$;
      end if
    end for
    $X_r(\vec{N}) \leftarrow N_r / \sum_{i=1}^{K} R'_{i,r}(\vec{N})$
  end for
  for $r \leftarrow 1$ to R **do**
    for $q \leftarrow 1$ to Q **do**
      $n^e_{q,r}(\vec{N}) \leftarrow X_r(\vec{N}) R'_{q,r}(\vec{N})$;
    end for
  end for
until $\boxed{\max_{q,r} |[n^e_{q,r}(\vec{N}) - n_{q,r}(\vec{N})] / n^e_{q,r}(\vec{N})|} < \epsilon$

In the above algorithm, $n_{q,r}(\vec{N})$ and $n^e_{q,r}(\vec{N})$ respectively represent the current and estimated queue lengths for queue q and class r with load intensity vector $\vec{N}$. $D_{q,r}$ denotes class r demands on queue q, and $R'_{q,r}(\vec{N})$ denotes class r residence time on queue q for the load intensity vector $\vec{N}$. Finally, X_r represents the class r throughput, ϵ represents the relative error tolerance, and Q, the number of queues. K_r gives the number of visitations of the r^{th} queue.

As we can see, the above algorithm can be parallelised by allocating each class to one of the processes. All the computations, except for the two boxed ones, can be performed simultaneously. The two boxed computations are: (1) summation of remote queue lengths, and (2) determination of the global maximum relative error, which ensures that all the processes terminate after executing the same number of iterations. The parallelised algorithm executed by a process is as follows:

$\vec{N} \leftarrow (N_1, \ldots, N_r, \ldots, N_R)$;
$r \leftarrow$ **process_rank**;
for $q \leftarrow 1$ to Q **do**
  $n^e_{q,r}(\vec{N}) \leftarrow N_r/K_r$;
end for
repeat
  for $q \leftarrow 1$ to Q **do**
    $n_{q,r}(\vec{N}) \leftarrow n^e_{q,r}(\vec{N})$;
  end for
  for $q \leftarrow 1$ to Q **do**
    $n_i(\vec{N} - \vec{1}_r) \leftarrow [N_r - 1] n_{q,r}(\vec{N})/N_r + \boxed{\sum_{t=1 \wedge t \neq r}^{R} n_{q,t}(\vec{N})}$;
    if *delay queue* **then**
      $R'_{q,r}(\vec{N}) \leftarrow D_{q,r}$;
    else
      $R'_{q,r}(\vec{N}) \leftarrow D_{q,r}[1 + n_q(\vec{N} - \vec{1}_r)]$;
    end if
  end for
  $X_r(\vec{N}) \leftarrow N_r / \sum_{i=1}^{K} R'_{i,r}(\vec{N})$
  for $q \leftarrow 1$ to Q **do**
    $n^e_{q,r}(\vec{N}) \leftarrow X_r(\vec{N}) R'_{q,r}(\vec{N})$;
  end for
until $\boxed{\max_{q,r} |[n^e_{q,r}(\vec{N}) - n_{q,r}(\vec{N})]/n^e_{q,r}(\vec{N})|} < \epsilon$

If we focus on the communications only, we can observe the communication pattern shown in Figure 6.1. Every iteration consists of queue length summation (first box), and maximum relative error determination (second box).

In addition to providing programmer control over the number of iterations, a set of classes can be allocated to a process, depending on the number of processes that are available for the computation. When the problem space contains a large number

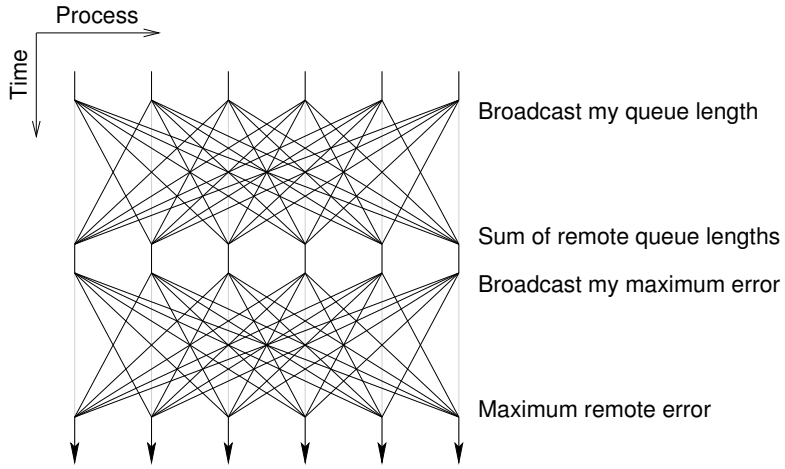


Figure 6.1: Communication pattern manifested by the parallelised mean value analysis algorithm, based on Schweitzer's sequential approximation algorithm.

of classes with large populations, each process can compute the local values for a class—simultaneously—when other processes are computing theirs. Compared to the pipeline parallelisation used by Gennaro and King, this algorithm is more efficient because all the processes can start, execute, and exit simultaneously. The other advantage is that each class can be allocated to a process, which does not happen in the pipeline approach because the problem space decomposition is based on the reference class population.

We shall now turn our attention to the implementation. Two approaches for implementing the above algorithm are discussed. The first one concerns implementations with MPI collective communications, the second with β-channels. To make the discussions both clearer and concise, we show in Figure 6.2 the common macros and variables that are used in all of the following implementations.

In Figure 6.1, we can see that the communication domains overlap. For example, the broadcast of the queue length on one process overlaps with the sum reductions on other processes. In the MPI implementation we must decompose this overlap, which presents us with a choice of three possible implementations using: (1) `MPI_Bcast()`, (2) `MPI_Reduce()`, and (3) `MPI_Alltoall()`. In fact, a mixture of these three collective communications can be adopted for the two separate communication patterns, as they both manifest the same data flow pattern. We shall now discuss these implementations.

```
1  #define val(V,X,Y) V[X * nqueue + Y]
   #define ptr(V,X,Y) &V[X * nqueue + Y]
3  int nclass; /* Number of classes. */
   int nqueue; /* Number of queues. */
5  float epsilon := 0.000005; /* Relative error tolerance. */
   int *qtype; /* Queue types. */
7  float *load_vect; /* Load intensity vector. */
   float *visit_vect; /* Queue visitation vector. */
9  float *throughput; /* Class throughput. */
   float *serv_demand; /* Service demand. */
11 float *qlen_est; /* Queue length estimate. */
   float *qlen; /* Current queue length estimate. */
13 float *res_time; /* Residence times. */
```

Figure 6.2: Common macros and variables used in the implementation of the parallelised mean value analysis algorithm. The first two macros are used to access individual elements in a one-dimensional representation of a two-dimensional array: the first expands to a variable, the second to a variable pointer.

Implementation with MPI_Bcast()

The implementation of the mean value analysis algorithm with the collective operation MPI_Bcast() is shown in Figure 6.3. At lines 13–17, we calculate the summation of all the remote queue lengths. As we can see, MPI_Bcast() is invoked nclass times, each representing a queue length broadcast from the current root. After every MPI_Bcast() invocation, the queue length received from the current broadcast root is added to the existing value of sum_qlens. The predicate ($k \neq i$) ensures that only the remote queue lengths are added (when the process is the root of the broadcast, its queue length should be ignored).

The termination condition is checked with isdone_bcast(). In this function, we perform an iterative broadcast, as above. However, instead of performing summation, we determine the maximum relative error (line 38–42). The parameters, i and size, respectively represent the rank of the calling process, and the total number of processes participating in the computation (equals the number of classes nclass, in this case).

Implementation with MPI_Reduce()

The implementation of the mean value analysis algorithm with the collective operation MPI_Reduce() is shown in Figure 6.4. Most of the implementation details remain the same, except for the summation of queue lengths, and maximum relative error determination. At lines 15–18, we perform the summation. We use MPI_Reduce() with the summation operator MPI_SUM. As this summation also in-

```
1  void mva_bcast ( void ) {
       int i, j, k;
3      float temp, one_less; /* Queue length with one less class r request. */
       float sum_qlens; /* Sum of the queue lengths excluding the current class. */
5      float sum_resis; /* Sum of the residence times for the current class. */
       MPI_Comm_rank (MPI_COMM_WORLD, &i);
7      for ( j := 0; j < nqueue; j++ ) /* Estimate queue lengths. */
           if ( val(serv_demand,i,j) > 0 ) val(qlen_est,i,j) := (float)load_vect[i]/visit_vect[i];
9      /* Parallel execution on each process. */
       do { for ( j := 0; j < nqueue; j++ ) val(qlen,i,j) := val(qlen_est,i,j);
11         for ( j := 0; j < nqueue; j++ ) {
               /* Queue length with one less class i request. */
13             for ( k := 0, sum_qlens := 0.0; k < nclass; k++ ) {
                   if ( k = i ) temp := val(qlen,i,j);
15                 MPI_Bcast (&temp, 1, MPI_FLOAT, k, MPI_COMM_WORLD);
                   if ( k ≠ i ) sum_qlens += temp;
17             }
               one_less := (load_vect[i] − 1)/load_vect[i]∗val(qlen,i,j) + sum_qlens;
19             /* Class i residence time at queue j. */
               if ( qtype[j] = DELAY ) val(res_time,i,j) := val(serv_demand,i,j);
21             else val(res_time,i,j) := val(serv_demand,i,j)∗(1.0 + one_less);
           }
23         for ( j := 0, sum_resis := 0.0; j < nqueue; j++ ) sum_resis += val(res_time,i,j);
           throughput[i] := load_vect[i]/sum_resis; /* Throughput for class i. */
25         /* Compute new estimates for the queue lengths. */
           for ( j := 0; j < nqueue; j++ ) val(qlen_est,i,j) := throughput[i]∗val(res_time,i,j);
27     } while ( ¬ isdone_bcast (i, nclass));
   }
29 int isdone_bcast ( int i, int size ) {
       int j, k;
31     float temp, error := 0.0; /* Relative error. */
       float lerror := 0.0; /* Local maximum relative error. */
33     float rerror; /* Maximum relative error from remote processes. */
       for ( j := 0; j < nqueue; j++ ) {
35         error := fabs (( val(qlen_est,i,j) − val(qlen,i,j))/val(qlen_est,i,j));
           if ( error > lerror ) lerror := error;
37     }
       for ( k := 0, rerror := 0.0; k < size; k++ ) {
39         if ( k = i ) temp := lerror;
            MPI_Bcast (&temp, 1, MPI_FLOAT, k, MPI_COMM_WORLD );
41         if ( k ≠ i ) if ( temp > rerror ) rerror := temp;
       }
43     if ( lerror > rerror ) return ( lerror < epsilon );
       else return ( rerror < epsilon );
45 }
```

Figure 6.3: Implementation of the mean value analysis algorithm with the collective operation MPI_Bcast(). The function isdone_bcast() checks the termination condition by using MPI_Bcast(). Lines 13–17 and 38–42 show the interesting parts, which are different from other implementations.

```
1  void mva_reduce ( void ) {
       int i, j, k;
3      float one_less; /* Queue length with one less class r request. */
       float sum_qlens; /* Sum of the queue lengths excluding the current class. */
5      float sum_resis; /* Sum of the residence times for the current class. */
       MPI_Comm_rank (MPI_COMM_WORLD, &i);
7      /* Estimate queue lengths. */
       for ( j := 0; j < nqueue; j++ )
9          if ( val(serv_demand,i,j) > 0 ) val(qlen_est,i,j) := (float)load_vect[i]/visit_vect[i];
       /* Parallel execution on each process. */
11     do {
           for ( j := 0; j < nqueue; j++ ) val(qlen,i,j) := val(qlen_est,i,j);
13         for ( j := 0; j < nqueue; j++ ) {
               /* Queue length with one less class i request. */
15             for ( k := 0; k < nclass; k++ )
                   MPI_Reduce (ptr(qlen,i,j), &sum_qlens, 1, MPI_FLOAT, MPI_SUM, k,
17                         MPI_COMM_WORLD);
               sum_qlens −= val(qlen,i,j);
19             one_less := (load_vect[i] − 1)/load_vect[i]∗val(qlen,i,j) + sum_qlens;
               /* Class i residence time at queue j. */
21             if ( qtype[j] = DELAY ) val(res_time,i,j) := val(serv_demand,i,j);
               else val(res_time,i,j) := val(serv_demand,i,j)∗(1.0 + one_less);
23         }
           /* Throughput for class i. */
25         for ( j := 0, sum_resis := 0.0; j < nqueue; j++ ) sum_resis += val(res_time,i,j);
           throughput[i] := load_vect[i]/sum_resis;
27         /* Compute new estimates for the queue lengths. */
           for ( j := 0; j < nqueue; j++ ) val(qlen_est,i,j) := throughput[i]∗val(res_time,i,j);
29     } while ( ¬ isdone_reduce (i, nclass));
   }
31 int isdone_reduce ( int i, int size ) {
       int j, k;
33     float error := 0.0; /* Relative error. */
       float lerror := 0.0; /* Local maximum relative error. */
35     float rerror; /* Maximum relative error from remote processes. */
       float temp;
37     for ( j := 0; j < nqueue; j++ ) {
           error := fabs (( val(qlen_est,i,j) − val(qlen,i,j))/val(qlen_est,i,j));
39         if ( error > lerror ) lerror := error;
       }
41     for (k := 0; k < size; k++)
           MPI_Reduce(&lerror, &rerror, 1, MPI_FLOAT, MPI_MAX, k,
43                 MPI_COMM_WORLD);
       return ( rerror < epsilon );
45 }
```

Figure 6.4: Implementation of the mean value analysis algorithm with the collective operation MPI_Reduce(). The function isdone_reduce() checks the termination condition by using MPI_Reduce(). Lines 15–18 and 41–43 show the interesting parts, which are different from other implementations.

```
1  void mva_all_to_all ( void ) {
       int i, j, k;
3      float *qlens, *lqlen, one_less; /* Queue length with one less class r request. */
       float sum_qlens; /* Sum of the queue lengths excluding the current class. */
5      float sum_resis; /* Sum of the residence times for the current class. */
       MPI_Comm_rank (MPI_COMM_WORLD, &i);
7      for ( j := 0; j < nqueue; j++ ) /* Estimate queue lengths. */
           if ( val(serv_demand,i,j) > 0 ) val(qlen_est,i,j) := (float)load_vect[i]/visit_vect[i];
9      /* Parallel execution on each process. */
       qlens := (float *) malloc (nclass*sizeof(float));
11     lqlen := (float *) malloc(nclass*sizeof(float));
       do { for ( j := 0; j < nqueue; j++ ) val(qlen,i,j) := val(qlen_est,i,j);
13         for ( j := 0; j < nqueue; j++ ) {
               /* Queue length with one less class i request. */
15             for (k := 0; k < nclass; k++) lqlen[k] := val(qlen,i,j);
               MPI_Alltoall (lqlen, 1, MPI_FLOAT, qlens, 1, MPI_FLOAT,
17                           MPI_COMM_WORLD);
               for ( k := 0, sum_qlens := 0.0; k < nclass; k++ )
19                 if ( k ≠ i ) sum_qlens += qlens[k];
               one_less := (load_vect[i] − 1)/load_vect[i]*val(qlen,i,j) + sum_qlens;
21             /* Class i residence time at queue j. */
               if ( qtype[j] = DELAY ) val(res_time,i,j) := val(serv_demand,i,j);
23             else val(res_time,i,j) := val(serv_demand,i,j)*(1.0 + one_less);
           }
25         for ( j := 0, sum_resis := 0.0; j < nqueue; j++ ) sum_resis += val(res_time,i,j);
           throughput[i] := load_vect[i]/sum_resis; /* Throughput for class i. */
27         /* Compute new estimates for the queue lengths. */
           for ( j := 0; j < nqueue; j++ ) val(qlen_est,i,j) := throughput[i]*val(res_time,i,j);
29     } while ( ¬ isdone_all_to_all (i, nclass)); free(qlens); free(lqlen);
   }
31 int isdone_all_to_all ( int i, int size ) {
       int j, k;
33     float error := 0.0; /* Relative error. */
       float lerror := 0.0; /* Local maximum relative error. */
35     float *temp, *rerrors; /* Relative errors from remote processes. */
       for ( j := 0; j < nqueue; j++ ) {
37         error := fabs (( val(qlen_est,i,j) − val(qlen,i,j))/val(qlen_est,i,j));
           if ( error > lerror ) lerror := error;
39     }
       rerrors := (float *) malloc (size*sizeof(float));
41     temp := (float *) malloc(size*sizeof(float));
       for ( k := 0; k < nclass; k++ ) temp[k] := lerror;
43     MPI_Alltoall (temp, 1, MPI_FLOAT, rerrors, 1, MPI_FLOAT, MPI_COMM_WORLD);
       for ( k := 1, lerror := rerrors[0]; k < nclass; k++ )
45         if (rerrors[k] > lerror) lerror := rerrors[k];
       free(rerrors); free(temp); return ( lerror < epsilon );
47 }
```

Figure 6.5: Implementation of the mean value analysis algorithm with the collective operation MPI_Alltoall(). The function isdone_alltoall() checks the termination condition by using MPI_Alltoall(). Lines 15–19 and 40–46 show the interesting parts, which are different from other implementations.

cludes the local queue length, this is subtracted to give the correct value (line 18). Note that MPI_Reduce() is invoked nclass times, each participating individually in the summation at one of the nclass processes.

The termination condition is checked with isdone_reduce(). Similar to the summation of queue lengths, this function also uses the collective operation MPI_Reduce(), however, the maximum value operator MPI_MAX is used instead of MPI_SUM (line 42–43). As the calculated maximum incorporates the local maximum relative error, we do not compare this value with the local value, contrary to what was done in the case of the MPI_Bcast() implementation at line 43.

Implementation with MPI_Alltoall()

The implementation of the mean value analysis algorithm with the collective operation MPI_Alltoall() is shown in Figure 6.5. Again, most of the implementation details remain the same, except for the calculation of the queue length summation, and maximum relative error determination. At lines 15–19, we perform the summation by using MPI_Alltoall(). Since this collective operation transfers data from any array of values to all of the remote processes, while also receiving new values from the remote processes, we allocate two arrays qlens and lqlen (line 10–11). The collective operation MPI_Alltoall() is invoked only once because when the call returns, qlens is filled with all the values required for the summation (including the local queue length set in lqlen). At lines 18–19, the summation is performed separately. Once more, we ignore the local queue length with the predicate (k $\neq$ i) (line 19).

The termination condition is checked with isdone_all_to_all(). Again, calculation of the global maximum relative error is similar to that of the queue length summation; however, the only difference occurs during the determination of the maximum value (lines 44–45), where we calculate the maximum value instead of calculating a summation.

Implementation with β-channels

In this section we discuss implementation of the mean value analysis algorithm with β-channels. Contrary to the above three implementations, only one is possible with β-channels. This implementation, shown in Figure 6.6, is based on the use of localised patterns visible to each of the processes. We can observe from Figure 6.1 that there are three localised communication patterns: (1) broadcasting the local queue length, or the local maximum relative error, to all the remote processes, (2) sum reduction of all the queue lengths received from remote processes, and (3) determination of the global maximum relative error.

```
1  void mva_bc ( void ) {
       int i := bc_rank, j;
3      float one_less; /* Queue length with one less class r request. */
       float sum_qlens; /* Sum of the queue lengths except for the current class. */
5      float sum_resis; /* Sum of the residence times for the current class. */
       bc_chan_t *src_data, *src_err, *sink;
7      for ( j := 0; j < nqueue; j++ ) /* Estimate queue lengths. */
           if ( val(serv_demand,i,j) > 0 ) val(qlen_est,i,j) := (float)load_vect[i]/visit_vect[i];
9      /* Create communication structures. */
       sink := bc_sink_create (bc_plist_xall, bc_float, 10, BC_ROLE_REPLICATE);
11     src_data := bc_src_create (bc_plist_xall, bc_float, BC_ROLE_REDUCE_SUM);
       src_err := bc_src_create (bc_plist_xall, bc_float, BC_ROLE_REDUCE_MAX);
13     /* Parallel execution on each process. */
       do { for ( j := 0; j < nqueue; j++ ) val(qlen,i,j) := val(qlen_est,i,j);
15          for ( j := 0; j < nqueue; j++ ) {
               /* Queue length with one less class i request. */
17             bc_put (sink, ptr(qlen,i,j), 1); bc_get (src_data, &sum_qlens, 1);
               one_less := (load_vect[i] - 1)/load_vect[i]*val(qlen,i,j) + sum_qlens;
19             /* Class i residence time at queue j. */
               if ( qtype[j] = DELAY ) val(res_time,i,j) := val(serv_demand,i,j);
21             else val(res_time,i,j) := val(serv_demand,i,j)*(1.0 + one_less);
           }
23         for ( j := 0, sum_resis := 0.0; j < nqueue; j++ ) sum_resis += val(res_time,i,j);
           throughput[i] := load_vect[i]/sum_resis; /* Throughput for class i. */
25         /* Compute new estimates for the queue lengths. */
           for ( j := 0; j < nqueue; j++ ) val(qlen_est,i,j) := throughput[i]*val(res_time,i,j);
27     } while ( ¬ isdone_bc (src_err, sink) );
       /* Destroy communication structure. */
29     bc_chan_destroy (src_data); bc_chan_destroy (src_err); bc_chan_destroy (sink);
   }
31 int isdone_bc ( bc_chan_t *src, bc_chan_t *sink ) {
       int i := bc_rank, j;
33     float lerror := 0.0; /* Local maximum relative error. */
       float rerror; /* Maximum relative error from remote processes. */
35     for (j := 0; j < nqueue; j++) {
           rerror := fabs((val(qlen_est,i,j) - val(qlen,i,j))/val(qlen_est,i,j));
37         if (rerror > lerror) lerror := rerror;
       }
39     bc_put (sink, &lerror, 1); bc_get (src, &rerror, 1);
       if (lerror > rerror) return (lerror < epsilon);
41     else return (rerror < epsilon);
   }
```

Figure 6.6: β-channel implementation of the mean value analysis algorithm. The communication structures created at lines 10–12 are used for communication of queue lengths, and maximum relative errors (lines 17 and 40). While sending maximum relative error, the isdone_bc() uses the same β-channel used for sending the queue length. However, it uses src_err while receiving the maximum 'remote' relative error. src_data is associated with the role BC_ROLE_REDUCE_SUM, while src_err is associated with BC_ROLE_REDUCE_MAX. The sink β-channel sink, on the other hand, is associated with BC_ROLE_REPLICATE.

Based on the two-phases application development process (see Section 4.1), we begin by defining the β-channels (lines 10–12) which correspond to the above three communication patterns. For this, two source β-channels, src_data and src_err, are created: each of them associated respectively with the roles BC_ROLE_REDUCE_SUM and BC_ROLE_REDUCE_MAX. We only need one sink β-channel as it can be used both for communicating the queue length, and the maximum relative error. This sink β-channel is associated with the BC_ROLE_REPLICATE role. We use the in-built process list, bc_plist_xall, to define the set of remote processes.

The three β-channels created in the communication structuring phase are then used during the communication activation phase (lines 17 and 39). Once the termination condition is satisfied, the β-channels are destroyed before returning (line 29).

Property	mpi approach	β-channel approach
Non-ambiguity	**Ambiguous:** Three implementations are possible. If we use different collective communications for queue length summation and maximum error determination, there are more possible implementations.	**Non-ambiguous:** Only one implementation is possible. Since the holistic pattern is abstracted in terms of the localised patterns, and there is only one way in which a process can represent its pattern, the runtime composition of these patterns also represents only one implementation of the holistic pattern.
Expressiveness	**Not expressive:** The patterns are not easily expressible (one of the reasons for the ambiguity). Adjustments have to be made so that the patterns fit the available interfaces. After implementation, information about the pattern is lost.	**Expressive:** The 'roles' describe exactly what the communication patterns mean. No information is lost during the implementation. A brief scrutiny of the β-channel creation code segments automatically reveals the communication patterns involved.
Uniformity	**Not uniform:** If we compare the three interfaces MPI_Bcast(), MPI_Reduce(), and MPI_Alltoall() in the previous sections, we can see the non-uniformity in the function prototypes—the varying number of parameters, and the parameter data types.	**Uniform:** Creation of the β-channels uses either of the two interfaces bc_src_create() or bc_sink_create(). The pattern is only defined by the different roles passed to these functions. During activations, no matter which pattern is associated with the β-channel, the following interfaces bc_put() (or bc_commit()), and bc_get() are required.
Extensibility	**Not easily extensible:** Although new collective communication interfaces can be introduced to express a communication pattern, management and usage of the interfaces becomes complicated because of the non-uniformity.	**Easily extensible:** By default, most of the patterns can be easily realised by combining the already existing 'roles'. When new patterns are necessary, all we need to do is introduce a 'role' which represents the pattern. This new addition can again be combined with already existing roles, therefore scaling the advantage of every new addition. Furthermore, the β-channel creation, activation, and destruction interfaces need not be changed.

Table 6.1: Comparison between the MPI approach and the β-channel approach in terms of non-ambiguity, expressiveness, uniformity, and extensibility.

6.1.1 Discussion on the qualitative properties

In Table 6.1 we summarise the qualitative evaluation of the β-channel approach, as compared to the MPI approach, based on programmability—non-ambiguity, expressiveness, uniformity, and extensibility. We shall now discuss these properties with examples.

Non-ambiguity. While discussing implementation of the mean value analysis (see page 135), we showed in Figure 6.1 the communication pattern manifested by the algorithm. From this algorithm, we could derive three different MPI implementations based on different combinations of collective communications. This ambiguity introduces complexity to the already complicated programming exercise by subjecting the programmer to a dilemma of choice (see page 4). In addition to the qualitative effects, the ambiguity could result in a performance portability problem (see page 4), which we demonstrate experimentally in Section 6.2.3. On the other hand, with the β-channel approach, we could implement the given communication pattern uniquely, as the process specific patterns can be mapped directly with the available roles (see lines 10–12 in Figure 6.6).

Expressiveness. Subsequent to the above discussion on non-ambiguity, we can observe that the MPI implementations require adjustment of the communication pattern to fit the interfaces. For example, at line 15 in Figure 6.3, in order to use the collective communication `MPI_Bcast()`, we had to sacrifice information (see page 5) on the sum reduction pattern, which could have been realised with `MPI_Reduce()`. Similarly, at lines 16–17 in Figure 6.4, to realise the sum reduction pattern using `MPI_Reduce()`, we had to sacrifice the information on the broadcast. We can make similar observations on the loss of structural information in the implementation with `MPI_Alltoall()`. In the β-channel implementation, we do not face this problem, as the communication patterns displayed by the algorithm (see Figure 6.1) can be expressed immediately, without sacrificing structural information (see lines 10–12 in Figure 6.6). Since the patterns are specified using β-channel roles, any MPI implementation of a given algorithm can be converted to a β-channel implementation, by analysing the communication pattern; whereas, converting a β-channel implementation to an MPI implementation would require adaptation of the communication pattern to fit the MPI interfaces.

Uniformity. As discussed in Section 1.2, the aim of the β-channel approach is to provide programming interfaces that are uniform in terms of the function prototype so that pattern integration does not require changes in the interfaces themselves. From the MPI implementations shown in Figure 6.3, Figure 6.4 and Figure 6.5, the interfaces change with the chosen communications pattern. These

patterns also have different function prototypes. For example, the MPI_Bcast() interface requires five parameters, while the MPI_Reduce() and MPI_Alltoall() interfaces both require seven parameters; with different parameter data types. These makes the MPI interfaces nonuniform, as compared to the β-channel interfaces where the patterns are encapsulated within the β-channels, leaving the activation interfaces, bc_put() and bc_get(), independent of the patterns. The uniformity of interfaces is relevant to the extension of the programming model which we discuss in the following paragraph.

Extensibility. In the previous paragraph, we discussed the MPI and β-channel interfaces in terms of uniformity, and we will now discuss how uniformity affects extensibility. In order to extend the programming interfaces with new communication patterns, the MPI approach requires the introduction of new interfaces. This is because communication patterns are associated with the interface name. By introducing new patterns, we also introduce new interfaces, which, as a result of the nonuniformity of MPI, results in the expansion of the interface set. An application programmer, using the MPI approach, is therefore required to acknowledge these new interfaces in addition to acknowledging the new patterns. With the β-channel approach, however, introducing a new pattern only means introducing a new role (as shown on page 69), without changing the activation interfaces, bc_put() and bc_get(), or the communication structuring interfaces, bc_src_create() and bc_sink_create().

6.2 Quantitative evaluation

In this section, we evaluate the β-channel interfaces quantitatively, through experimentation. In Section 6.2.1, we compare the performances of point-to-point communication interfaces. In Section 6.2.2, we compare the performances of collective communication interfaces.

The setup for the experiments is a 64 node beowulf cluster consisting of Dell OptiPlex GX240 workstations, each with a 1.8-GHz Pentium 4processor with 256MB PC133 SDRAM, connected through two 100Mb Ethernet networks.

The environment is based on the LINUX operating system (Fedora core 3 release), version 2.6.12-1.1372_FC3. We use GCC, version 3.4.4 20050721 for compilation. The runtime system for the β-channel programming model is implemented with native TCP/IP sockets [94]. The multi-threading is done with POSIX threads [25]. For the MPI implementation, we choose LAM-MPI [24], version 7.1.1.

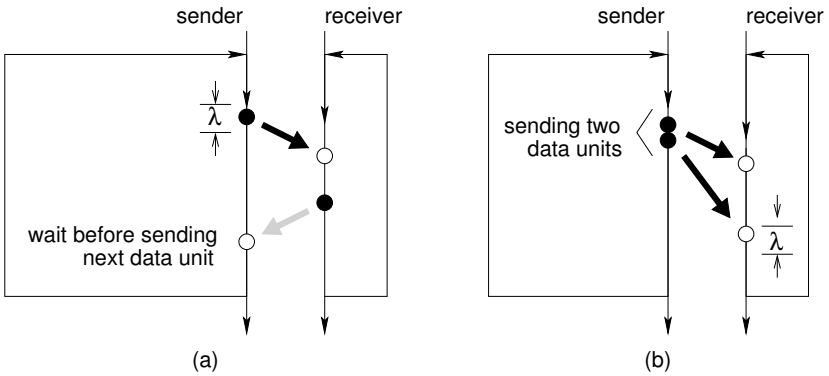


Figure 6.7: Evaluation method for interface latency, represented by λ. Each • represents a data unit that has been sent, and each ○ represents a data unit that has been received. (a) Evaluation of *sender latency*: the feint arrow represents acknowledgement message. This ensures that the latency values evaluated in each of the iterations are statistically independent. (b) Evaluation of *receiver latency*: the evaluation takes place in the second receive invocation, while the first receive invocation ensures that messages are ready when the second receive is invoked. Note that the sender sends two data units with a single interface call, and this ensures that the second receive will not be delayed because of data unavailability on the sender.

6.2.1 Point-to-point performance

In this section we discuss the experimental results for the point-to-point communication interfaces. The evaluation is performed as shown in Figure 6.7, where λ represents the interface latency. We classify the latencies into two types: sender latency, and receiver latency.

Sender latency. The *sender latency* is defined as the time (in time units) which a sender process spends when a message is being sent, or put into an auxilliary location (for example, a message buffer) to be retrieved later by the runtime system. For each message size, n (= 1001) latency evaluations are performed within the loop. The statistical median of the n latencies is then chosen as the latency for that message size.

During an iteration, the sender process sends a message, and waits for an acknowledgement message from the receiver (shown with feint arrow in Figure 6.7.a), before continuation. This acknowledgement message ensures that the n latencies resulting from the evaluation loop are statistically independent. For blocking send, this may not be necessary if the receiver loop guarantees that messages are accepted as soon as they are available (for example, a dedicated receiver). For buffered mode communications, however, we must ensure that the sender process is not sending

(a)

(b)

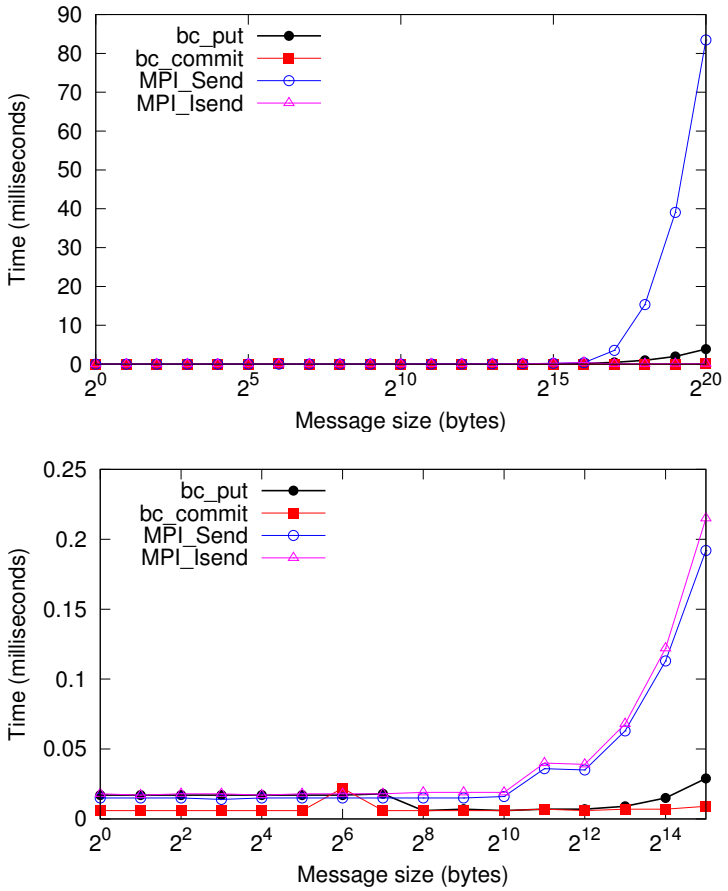


Figure 6.8: Comparison of sender latencies for bc_put(), bc_commit(), MPI_Send(), and MPI_Isend(). (a) Sender latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

more messages than the receiver can accept, filling up the buffer too soon—in such cases, the statistical independence is not preserved because the sender process will have to wait for existing messages to be removed from the buffer before sending a new message for the next evaluation.

In Figure 6.8, we compare the latencies for bc_put(), which incur intermediate memory copy; bc_commit(), which does not incur intermediate memory copy; MPI_Send(), blocking send; and MPI_Isend(), non-blocking send. Contrary to the popular belief that message buffering is a serious performance bottleneck, the evaluation of the sender latency shows that it is not always the case. Compared to the performance of MPI_Send(), the performance of bc_put() displays a remarkable improvement. This improvement affects the overall performance of the application program using these interfaces because the sender process does not have to wait for the receiver processes. Given that there is a large set of tasks, faster processes can

(a)

(b)

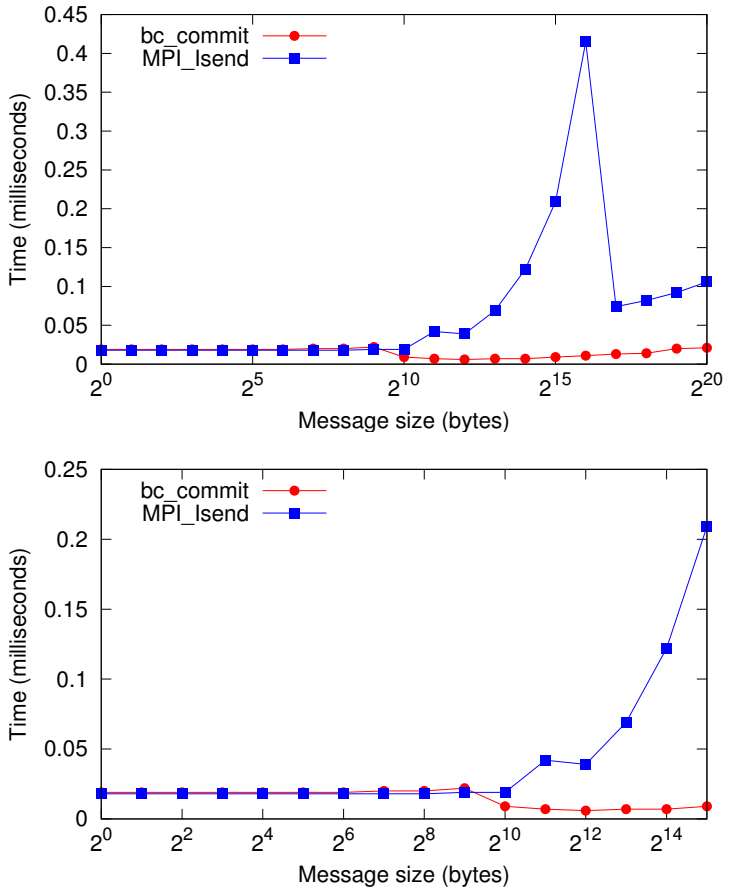


Figure 6.9: Comparison of sender latencies for asynchronous interfaces bc_commit() and MPI_Isend().(a) Sender latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

therefore continue with further computations—greedily—without being impeded by the slower processes.

If we compare the asynchronous mode communication interfaces only, as shown in Figure 6.9, we can observe that there is a significant difference between performances of the two interfaces. Given that bc_commit() does not require a program to check if message transfers are successful, this significant performance improvement suggests that the *commit* based interface is a better option for send-and-forget type communications.

Further observation reveals that, although MPI_Isend() is supposed to return immediately after making a service request from the MPI runtime system, in the current MPI implementation, messages of size $\leqslant 2^{16}$ are buffered. The plot, on the other hand, also shows that the latency for the bc_commit() interface is almost independent of the message size; and is smaller than the MPI_Isend() latency. This proves that the

(a)

(b)

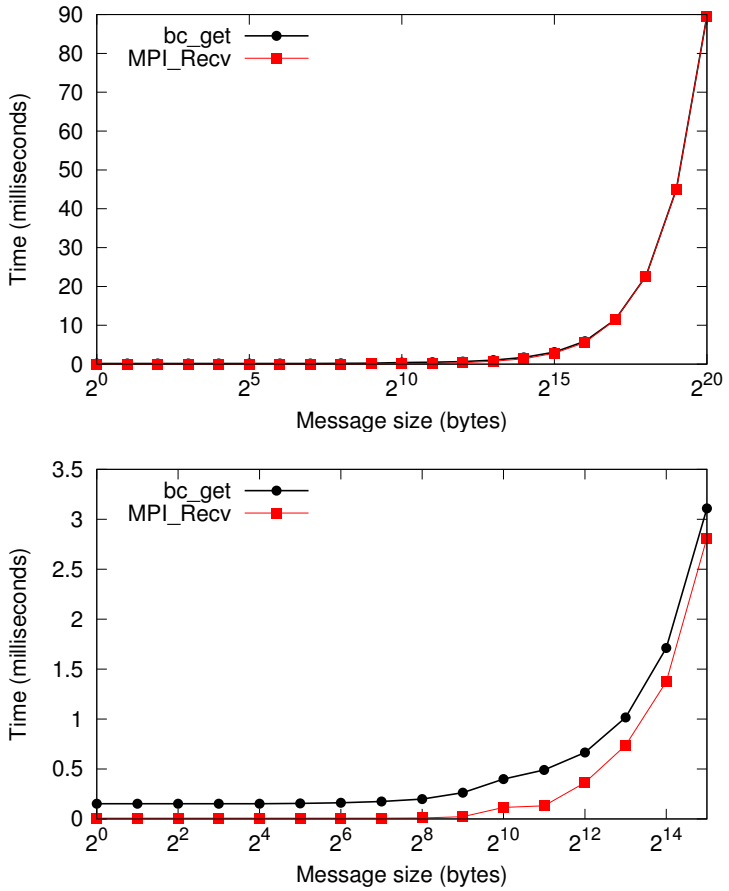


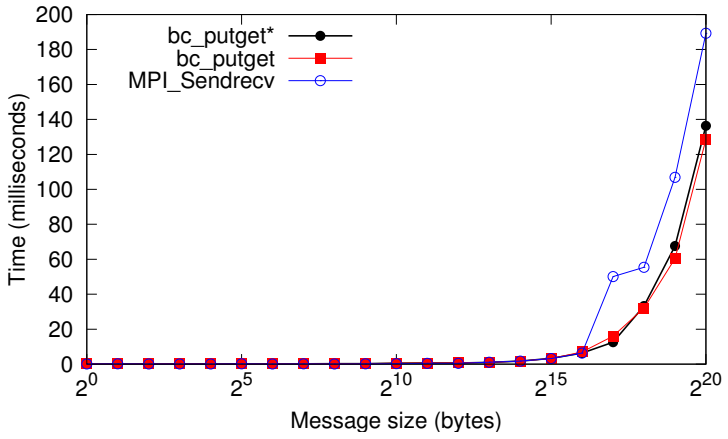
Figure 6.10: Comparison of receiver latencies for bc_get() and MPI_Recv(). (a) Receiver latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

algorithms for message buffering without intermediate memory copy, suggested in Section 5.5.1, are a valid, and more efficient alternative to those of the MPI_Isend() interface. This reinforces our argument that application programs where sender processes do not reuse sent data can achieve a significant performance boost by adopting a commit based interface.

Receiver latency. The *receiver latency* is defined as the time (in time units) spent by a receiver process on receiving messages from a remote process (or processes in the case of collective communications). If receiving a message requires sending a data transfer request to the remote processes, the waiting time incurred in such situations is also included in the evaluated latency.

The receiver latency is evaluated as shown in Figure 6.7.b. λ represents the receiver latency being evaluated. Similar to the evaluation of sender latency, for

(a)



(b)

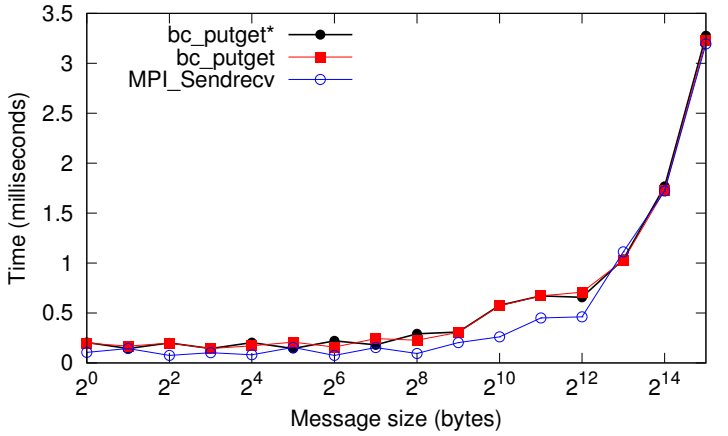


Figure 6.11: Lock-step exchange represents communications where data are exchanged between two processes. MPI provides the interface MPI_Sendrecv() for performing such communications. We compare this interface to the β-channel interfaces. The difference between the β-channel interfaces is that the put_get uses bc_put() while commit_get uses bc_commit() for sending messages.

each message size, n (= 1001) latency evaluations are performed within a loop. The statistical median of the n latencies is then chosen as the value of the receiver latency for that message size.

A sender process sends two consecutive, unique messages, of which the second message is used for the evaluation. Receiving the first message ensures that the second message to be received for the evaluation is available when the receive interface is invoked. The reason for this setup is that receiver latency should not be affected by the waiting time that might be incurred if the messages were not ready for retrieval. Such a situation might occur if, for example, the receiver process invoked the receive interface well before the messages were sent by the sender process. By using the first message, statistical independence is hence preserved because the sender process can

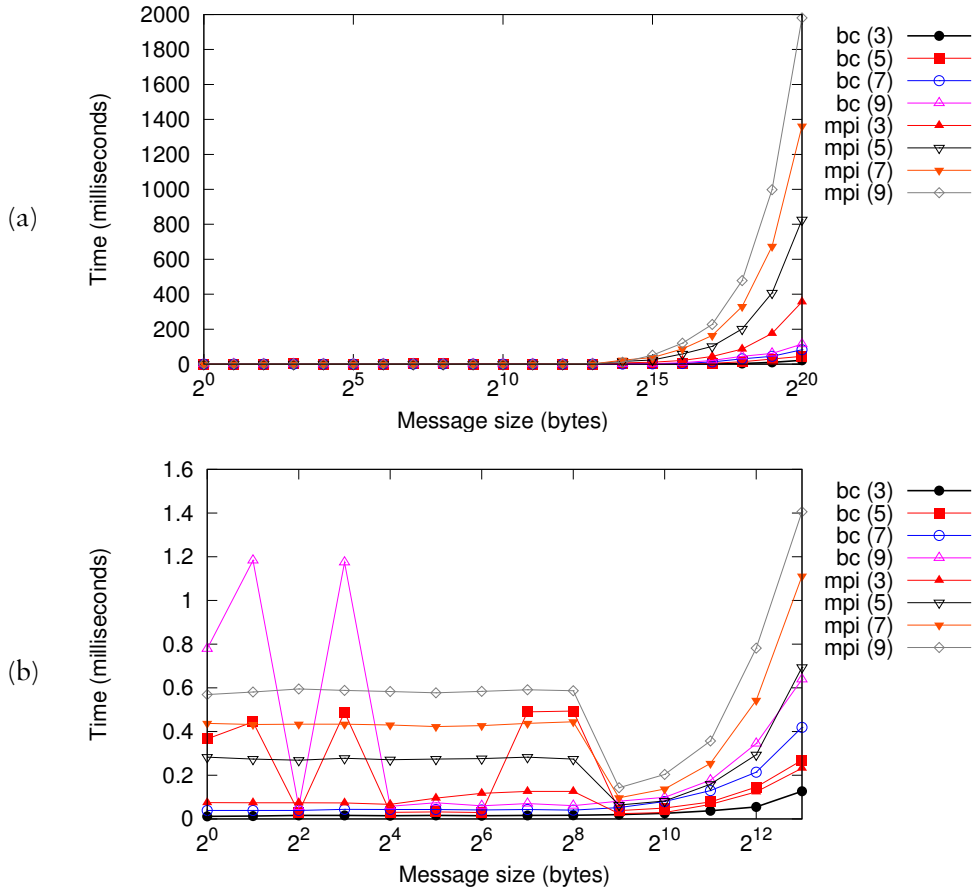


Figure 6.12: Performance of the data scattering interfaces, MPI_Scatter() and bc_put(). (a) Sender latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

always continue sending messages without having to wait for the receiver process, satisfying the condition for data availability in all of the following iterations.

As one might expect, the performance for bc_get(), as shown in Figure 6.10, is slightly inferior to that of MPI_Recv() for small messages. The reason for this inferiority is the asynchronous *rendezvous* communication protocol (see Section 5.4.3) that is used by the β-channel runtime system. As the protocol requires a service request to be sent to the remote process before receiving a message, this contributes to the additional communication costs. It should be noted, however, that the performance improves when larger messages are being received. The reason being that, as the size of the message increases, the communication costs due to the service request become negligible compared to the size of the actual message that is received. This compensates for the extra communication costs because in real applications, messages are not usually communicated very often; and if communicated, the com-

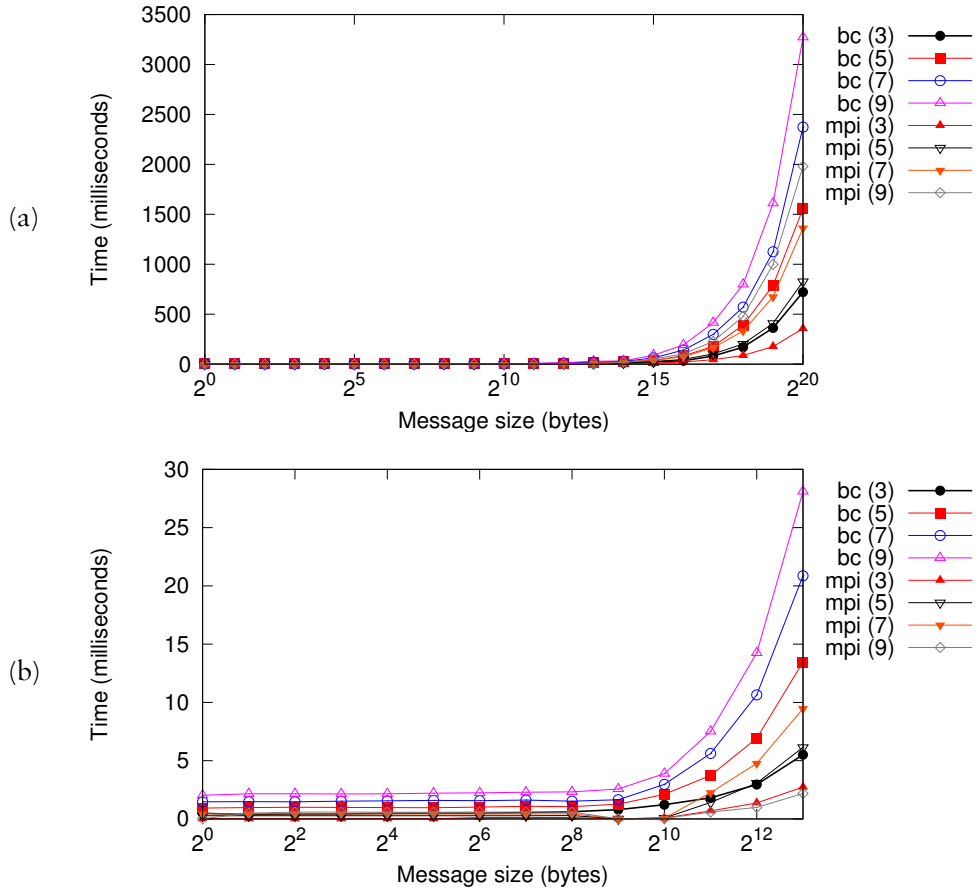


Figure 6.13: Performance of the data receiving interfaces, MPI_Scatter() and bc_get(), during *contention*. (a) Receiver latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

munications transfer large data sets. Further to this argument, we have noted at the end of Section 5.4.3 several conditions that will allow improvement of the β-channel runtime implementation, so that the cost for communicating a service request can be reduced to a satisfactory low value.

Lock-step exchange. Lock-step exchange represents communications where data are exchanged between two processes. They often appear in parallel algorithms, as we have seen in the odd-even transposition sorting algorithm, and the fast Fourier transform. While performing such communications with the interface pair, MPI_Send() and MPI_Recv(), there is a possibility of deadlocks if the interfaces are not ordered properly [93, page 33]. It is often necessary to use non-blocking interfaces, such as MPI_Isend(). To simplify such concerns, the MPI standard provides the interface MPI_Sendrecv() for performing such communications.

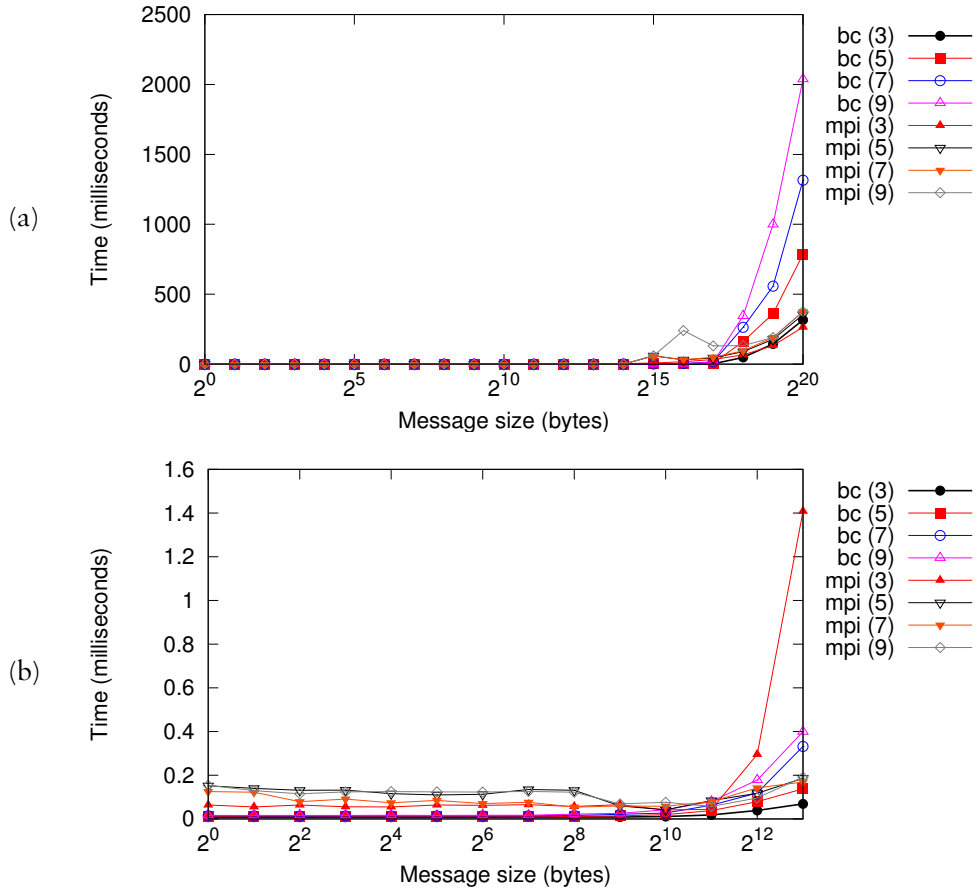


Figure 6.14: Performance of the data broadcasting interfaces, MPI_Bcast() and bc_put(). (a) Sender latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

We shall now compare the performance of MPI_Sendrecv() to that of the β-channel interface pairs, bc_put() and bc_get(), which uses the memory copy version; and bc_commit() and bc_get() which uses the commit based interfaces. Also, since the β-channel runtime system uses message buffers as an integral part of the communications, both processes can invoke bc_put() and bc_get() in the same order without causing a *deadlock*. The evaluated performance results are shown in Figure 6.11. We can observe that the performance of the β-channel interfaces perform slightly better than the MPI interface, MPI_Sendrecv(). We attribute this improvement to the simultaneous execution of the bc_put() and bc_get() interfaces on both processes. As for the two β-channel implementations, there is very little performance difference. We believe that this is because the time taken to receive data with a bc_get() decides the final result of the total latency.

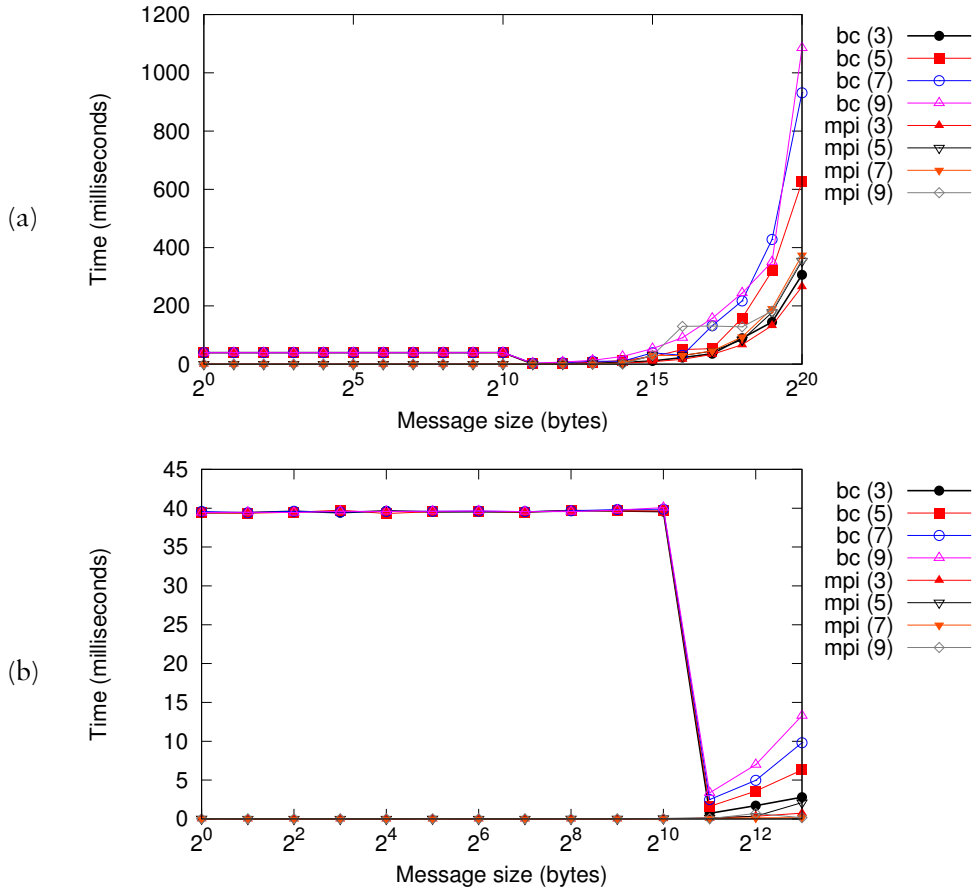


Figure 6.15: Performance of the data receiving interfaces, MPI_Bcast() and bc_get(), during *contention*. (a) Receiver latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

6.2.2 Collective performance

We shall now turn our attention to the performance of collective communication interfaces. Every collective communication interface is evaluated on 5, 10, 15 and 20 nodes of the beowulf cluster.

Scattering of data. The performances of the scattering interfaces are evaluated both on the sender and the receivers. The performance of the sender interface is shown in Figure 6.12, and the performance of the receiver interface 'during contention' is shown in Figure 6.13. Although both performance results are the same for the MPI interfaces (as a result of the barrier synchronisations), the results are not the same for the β-channel interfaces because there are no barrier synchronisations involved—each of the sender and receiver interfaces is invoked asynchronously.

As we can observe in Figure 6.12, the β-channel interface for scattering data

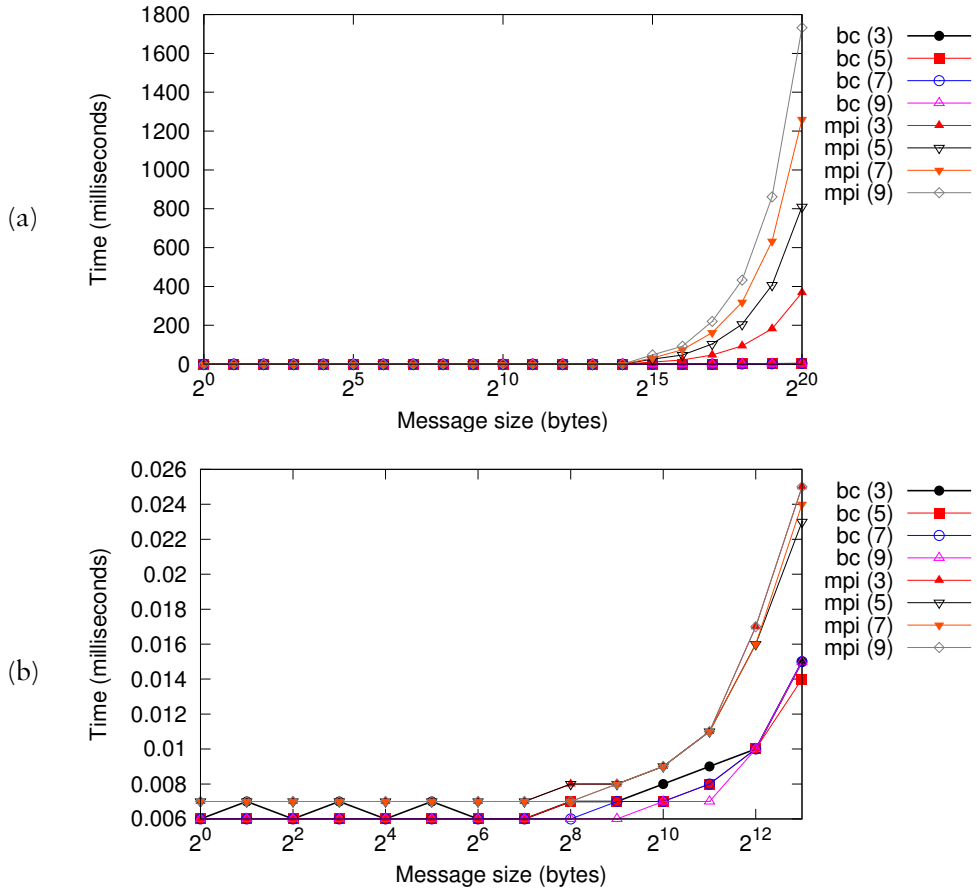


Figure 6.16: Performance of the data sending interfaces, MPI_Gather() and bc_put(). (a) Sender latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

from a root process performs better than the MPI interface. There is a difference of about 2 seconds for large data sets (2^{20} bytes), and about 500 microseconds for smaller data sets ($< 2^9$).

During contention, as we can observe in Figure 6.13, the β-channel interface for receiving data from the root process does not perform better than the MPI interface. There is difference of about 1.5 seconds for large data sets (2^{20} bytes), and about 250 microseconds for smaller data sets ($< 2^9$). This is because, when all the processes are competing for data at the root process, all the processes other than the one currently receiving data are waiting for the root process to respond with the necessary data.

However, when there is no contention for data on the root process, the performance of the β-channel receiver interface, bc_get(), remains the same, as shown in Figure 6.10. This means a saving of about 1.91 seconds for large data sets (2^{20} bytes).

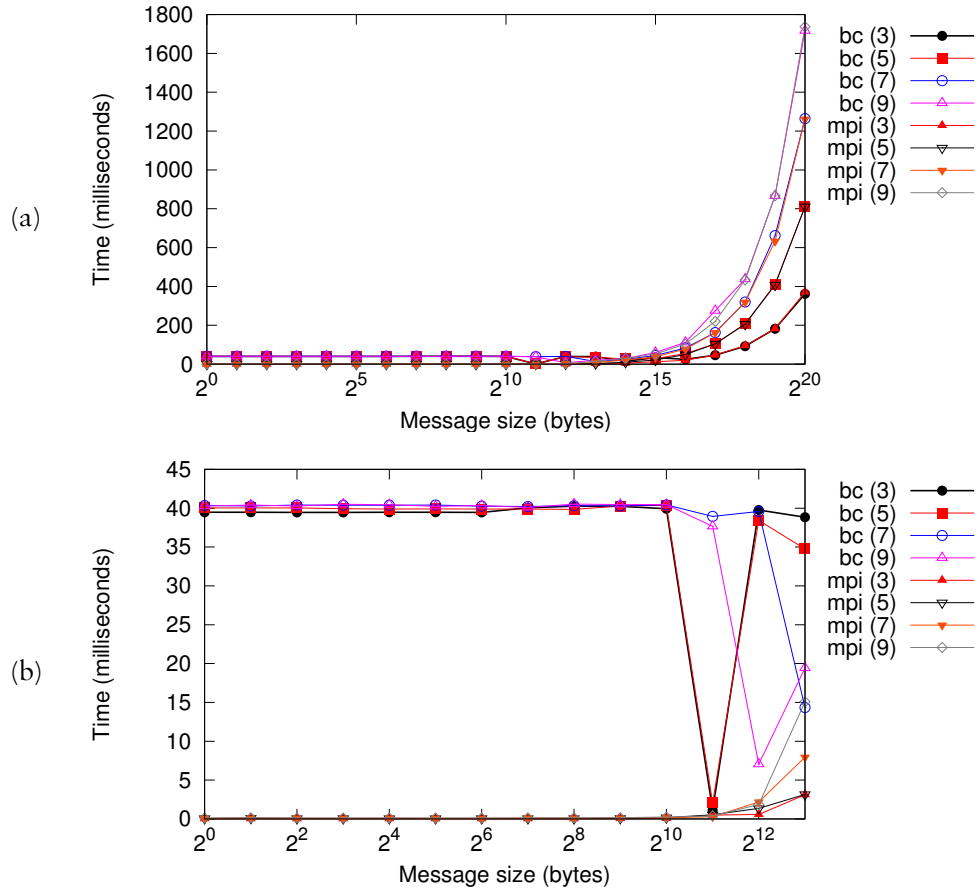


Figure 6.17: Performance of the data receiving interfaces, MPI_Gather() and bc_get(). (a) Receiver latency in milliseconds, against message size in bytes. (b) Zoom of (a) showing latency for smaller messages ($\leqslant 2^{15}$).

Data broadcast. The performances of the broadcasting interfaces are evaluated both on the sender and the receivers. The performance of the sender interface is shown in Figure 6.14, and the performance of the receiver interface 'during contention' is shown in Figure 6.15. Although both performance results are the same for the MPI interfaces (as a result of the barrier synchronisations), the results are again not the same for the β-channel interfaces because there are no barrier synchronisations involved—each of the sender and receiver interfaces is invoked asynchronously.

As we can observe in Figure 6.14, the β-channel interface for broadcasting data from a root process performs better than the MPI interface. There is a difference of about 275 milliseconds for large data sets (2^{20} bytes), and about 100 microseconds for smaller data sets ($< 2^9$).

During contention, as we can observe in Figure 6.15, the β-channel interface for receiving data from the root process does not perform as well as the MPI inter-

face. There is a difference of about 2.8 seconds for large data sets (2^{20} bytes), and about 250 microseconds for smaller data sets ($< 2^9$). This is because, when all the processes are competing for data at the root process, all the processes other than the one currently receiving data are waiting for the root process to respond with the necessary data, as was the case with scattering of data.

However, when there is no contention for data on the root process, the performance of the β-channel receiver interface, `bc_get()`, remains the same, as shown in Figure 6.10. This means a saving of about 242 milliseconds for large data sets.

Gathering of data. The performances of the data gathering interfaces are evaluated both on the senders and the receiver. The performance of the sender interfaces is shown in Figure 6.16, and the performance of the receiver interface is shown in Figure 6.17. Although, both performance results are the same for the MPI interfaces (as a result of the barrier synchronisations), the results are again not the same for the β-channel interfaces because there are no barrier synchronisations involved—each of the sender and receiver interfaces is invoked asynchronously.

As we can observe in Figure 6.16, the β-channel interface for sending data to the root process performs better than the MPI interface. There is a saving of about 1.7 seconds for large data sets ($\geqslant 2^9$ bytes), and about 7 microseconds for smaller data sets ($< 2^9$ bytes). The performance of the receiving interface, on the other hand, remains comparable to that of the MPI, as shown in Figure 6.17. The case of data contention on the sender does not apply in this case because all of the sender processes only perform a point-to-point communication.

Discussion. From the experimental results, we can see that the latencies of the β-channel interfaces for sending messages are relatively smaller than those of the MPI interfaces, and significantly smaller than the MPI interfaces for larger messages (see Figure 6.8). We see an improvement of about 87.5% for large messages (2^{15} bytes). With interface optimisations for *send-and-forget* type communications, the improvement is more significant, since the experimental results show that the sender latency is almost independent of the message sizes (see Figure 6.9).

As discussed previously (see Section 5.4.3), the asynchronous *rendezvous* communication protocol is receiver initiated; hence, the latencies of the interfaces for receiving data include the latency for sending the data transfer request. From the experimental results, we can observe that this is reflected for smaller messages. However, when the messages being received are large, the overhead for communicating the data transfer request becomes negligible (see Figure 6.10), therefore the performance almost equals that of MPI interfaces. Since receiver initiated communications increase the asynchrony between the communicating processes, we believe

that the degradation demonstrated by the receiver latency can be tolerated for small messages. As we can observe in the performance of lock-step exchange communications, the degradation resulting from the communication of the data transfer request is compensated for by the increase in asynchrony between the communicating processes.

For the collective communications, the performance of the β-channel interfaces remains the same when there is no contention for data among multiple processes. This means that, because of the asynchronous *rendezvous* communication protocol, each process can be treated independent of the other processes as long as they do not initiate communications at the same time. For situations where multiple processes are contending for data from a sender process, there is some degradation in performance among the contending processes since only one data serving thread on the sender process can serve a receiver process.

When comparing performance, we should account for the fact that the MPI implementation being used is a mature and portable system. Since the current implementation of the β-channel runtime system is a prototype, we should account for the features and overheads incorporated, or excluded, in either system. The performance results should therefore be considered as guidelines for further improvements to the β-channel runtime system.

6.2.3 Performance of the mean value analysis algorithm

In this section, we provide a macro-benchmark. We compare performances of the three MPI implementations of the mean value analysis, to that of the β-channel implementation (see Section 6.1). The performance results are shown in Figure 6.18.

Each of the four implementations of the mean value analysis algorithm is executed on 10 processes. The queueing network to be solved is initialised with 10 classes, so that each class can be assigned to a separate process. We perform five experiments by changing the class population from 64 to 1024, increasing the population size by two folds. As we can see, the performance of the β-channel implementation surpasses that of the two MPI implementations based on the collective operations, MPI_Bcast() and MPI_Reduce(). However, the performance of the β-channel implementation remains largely comparable to that of the MPI_Alltoall() implementation. The significant difference in the performance of the implementations using MPI_Bcast() and MPI_Reduce() as compared to the MPI_Alltoall() and the β-channel implementations is mainly due to the multiple barrier synchronisations that are required in the former two implementations.

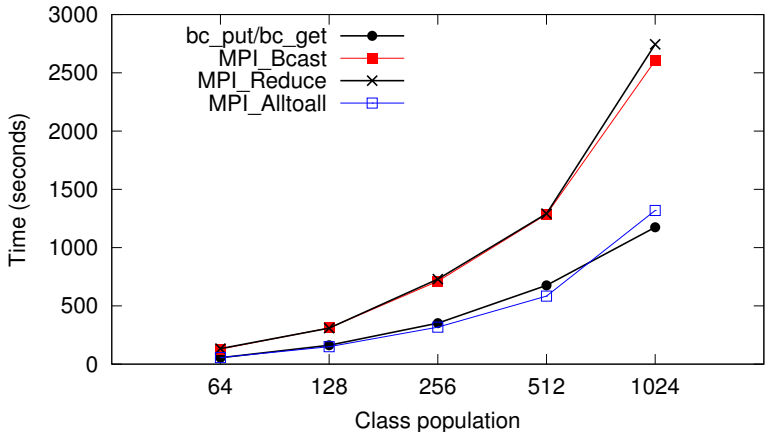


Figure 6.18: Performance comparisons for the mean value analysis algorithm implementation. Four different implementations based on the interfaces, MPI_Bcast(), MPI_Reduce(), MPI_Alltoall(), and bc_put()/bc_get(), are compared. The graph shows the execution time (in seconds) against the class population for a 10 class queueing network, when executed on 10 processes (with each class assigned to a different process). We vary the population size four times from 64 to 1024.

6.3 Summary

In this chapter, we have evaluated the β-channel programming model both qualitatively and quantitatively. We have discussed the qualitative advantages by analysing the parallelisation and implementation of Schwitzer's approximation based mean value analysis algorithm (see Section 6.1). We saw that three implementations were possible when MPI collective operations were used, as compared to a single implementation with the β-channel interfaces (see Section 6.1.1). We also saw that it was simpler to express the communication patterns when β-channels were used, instead of attempting to fit the communication patterns into the set of available MPI interfaces. We have compared these four implementations on a beowulf cluster and observed that the β-channel programming model indeed performed better than two of the above implementations, while remaining close in performance level to the other (see Figure 6.18). We have also provided performance comparisons for the point-to-point and collective communication interfaces, and noted that application programmers can take advantage of the asynchronous interfaces provided by the β-channel interfaces, instead of utilising MPI collective operations which often require barrier synchronisation including all of the processes in the group (see Section 6.2.3).

CHAPTER 7

Conclusion

In this dissertation we have introduced a new approach for message passing programming. Our approach is based on the thesis that communication patterns manifested by parallel algorithms are best abstracted as the runtime composition of process specific localised communication patterns. By avoiding the notion of a 'process group', we have resolved some of the programming issues that are inherent in process group based models, namely: *ambiguity* and *loss of structural information problem*, *choice dilemma* and *performance portability problem*, and finally the *problem of redundant acknowledgement*. We have also shown how the programming model defined by our approach is non-ambiguous, uniform, expressive and extensible. Furthermore, we have discussed several practical advantages—automatic overlapping of computations and communications, avoiding intermediate memory copy in send-and-forget type communications—which a programmer can exploit without understanding the internal implementation details of the runtime system. In this chapter we summarise the contributions made by this dissertation, and discuss potential avenues for further research.

7.1 Summary

In Chapter 1, we began with a general introduction on why abstraction models are necessary for message passing programming. In Section 1.1, we discussed the programming issues that are inherent in 'process group' based abstraction models (see pages 4–5. This provided the motivation for re-analysing existing concepts of a communication pattern, which led to our thesis that holistic communication patterns manifested by a parallel algorithm are best abstracted as the implied runtime composition of process specific localised communication patterns (see Section 1.2). This thesis allowed us to define a communication pattern while avoiding the notion of a 'process group'. In this section, we also defined the qualitative properties which we use to compare our approach with related systems (see page 6). The remainder of this chapter summarised the contributions, and provided an outline of the organisation of the remaining chapters of the dissertation; followed by a description of the mathematical and pseudo-code conventions.

In Chapter 2, we discussed the scope of this dissertation (see Section 2.1), and explored existing systems and approaches to message passing programming (see Section 2.2). We then described, in detail, the objectives of this dissertation (see Section 2.3). This was followed by a discussion of the approach we used to achieve the objectives, based on guidelines provided by works on the psychology of programming (see Section 2.4).

In Chapter 3 we began by re-analysing the concept of a communication pattern (see Section 3.1). Based on this analysis and our thesis, we developed the β-channel abstraction model. The abstraction model is based on the already established concept of a control flow graph (see Section 3.2); which we enhance for pattern integration by introducing new concepts. We first defined the concept of a *dependency point* (see Section 3.3) which represents the nodes in the control flow graph that produce data for, or consume data from, a remote process. To provide a framework for pattern integration, we defined the concept of a *dependency class* (see Section 3.4), which provides a logical grouping of dependency points. By using these dependency classes as the basis, we defined the concept of a *role* (see Section 3.6), which associates a process specific pattern to the dependency class. This approach allows communication activation interfaces to change functionality during activation (see Section 3.5), depending on the role of the dependency class. Finally, to simplify translation from the abstraction model to the programming model, we defined the concept of a *communication structure* (see Section 3.7), which in essence captures the communications, the patterns and any specialised communication properties,

such as message buffers and data types, that are required for a successful communication between processes. These communication structures are expressed in the application program as a set of data structures defined as *branching channels* (or β-channels) (see Section 3.8). In this chapter we discussed resolution of the programming issues raised in Section 1.1 (see page 33), and provided an outline of the advantages this approach has to offer (see Section 3.9).

While Chapter 3 was concerned with the development of a theoretically sound abstraction model, Chapter 4 concentrated on the practical aspects. In this chapter, we discussed the application development phases—*communication structuring phase* and *communication activation phase*—which allow separation of concerns (see Section 4.1). We then described the application programming interfaces (see Section 4.2) which correspond directly to the abstraction concepts introduced in Chapter 3. To demonstrate the simplicity of programming, we discussed implementations of five non-trivial parallel algorithms, in each case emphasising how the diverse communication patterns are captured by the β-channels (see Section 4.3). In the final part of this chapter, we discussed the relationship between the β-channel programming model and skeletal parallel programming (see Section 4.4), and suggested approaches which may improve the implementation and deployment of algorithmic skeletons.

In Chapter 5 we discussed the implementation details that are relevant to the understanding of the runtime system. We gave an overview of the design decisions (see Section 5.1), and discussed the internal details of the multi-threaded runtime system which overlaps computations and communications by introducing different types of threads within a process (see Section 5.2). In this section, we described the functional units of the runtime system. In Section 5.3, we discussed how the communication structures are realised at runtime, and highlighted issues that are related to the 'planarity condition'. In Section 5.4, we described the asynchronous *rendezvous* communication protocol, and contrasted this with the *split-phase asynchronous communication protocol*. This chapter concluded with a discussion of the low-level implementation details involved in the integration of programmer definable message buffers into the runtime system (see Section 5.5).

Finally, in Chapter 6 we evaluated the β-channel approach both qualitatively (see Section 6.1) and quantitatively (see Section 6.2). The evaluation was based on implementations of the mean value analysis algorithm, using both MPI and β-channel interfaces. To show the improvements in performance, we provided both micro- (see Section 6.2.1 and Section 6.2.2) and macro-benchmark (see Section 6.2.3) results. The experimental results showed that the β-channel interfaces had lower latency

than MPI interfaces for large messages during less contention; while demonstrating comparable performances during contention. It was shown that, by using the performance optimisation for *send-and-forget* type communications (see Section 5.5.1), applications could improve performance significantly, since the latency for the *commit* based interfaces has been shown to be almost independent of the message size being transferred (see Figure 6.9).

In conclusion, this dissertation has introduced a new way of understanding and capturing communication patterns. Our model—which is based on the thesis that holistic communication patterns are best abstracted as an implied runtime composition of process specific localised communication patterns—has resolved some of the subtle programming issues that are related to 'process group' based message passing models. By implementing several algorithms with diverse communication patterns, we have demonstrated that the β-channel programming model is non-ambiguous, uniform, expressive and extensible. Finally, through experimental results, we have shown that the performance of message passing applications can be improved by exploiting several features of the β-channel approach, for example, specialised message buffering, avoiding intermediate memory copy during buffering, single-phase asynchronous interfaces, and the automatic overlapping of computations and communications through the multi-threaded runtime system.

7.2 Further research

In this section we discuss possible avenues for further research.

Application to Grid environment. In Section 1.1, we discussed the problem of redundant acknowledgement, which is inherent in 'process group' models. In order to discuss a practical concern related to this issue, let us assume a dynamic environment where processes come and go, for example a Grid environment. In such an environment, let us execute an implementation of the first decomposition of Example 1.1.1; and focus on the scattering of data from the accountant. If dynamic nodes on the Grid represent student processes, imagine a situation where students enter and leave the system, as in a University. As collective communications are based on process groups, every student who enters or leaves this system should be acknowledged by all the other entities currently existing in the system. Hence, if we start with the situation contemplated in Example 1.1.1, and another student, say S_0, enters the system, then the process groups on A, T and S should be changed accordingly to reflect this new entry. However, these changes are not practically necessary for S and T because they do not communicate with S_0. This means, therefore, that

with process group models, localised changes affect the whole system: complicating the management of the Grid.

Since the β-channel approach resolves the problem of redundant acknowledgement, processes executing a β-channel application program only acknowledge the processes with which they directly share a communication link. From this, we can observe that the β-channel approach provides an interesting avenue for managing the Grid. In the current β-channel model, certain enhancements should be made to the ordering of the processes in the ensemble. This is necessary because, for some of the process lists (for example bc_plist_xall) the ordering does not matter, and therefore the insertion or removal of a process can be managed fairly easily; however, for other process lists (for example bc_plist_succ) the ordering of the processes is directly relevant to the management of the Grid. The question is how do we assign the ranks in such cases? Some interesting works in this direction are [95, 98].

Removing the planarity condition. Another avenue for research is the exploration of a new approach to internal tag assignment policy. We acknowledge that the 'planarity condition' can become a deterrent to the adoption of the β-channel approach. It would therefore be interesting to find an approach which removes the planarity condition, yet provides us with the same results as before. It is important to note here that any new tag assignment policy should also be completely asynchronous, without the need for global communications. In essence, what we need is a tag assignment policy that can be carried out statically, based on the compile-time information which we have on the communication structures. One approach which seems to show promise is the 'name' based tag assignment policy. In CSP [62], for example, the name of the channel defines the sink-to-source link. We can therefore devise such a scheme for the β-channel approach. One approach would be to use the C preprocessor macro; for example, the macro create_sink() which is defined as:

```
#define create_sink(name,plist,buffer,role) \
        name = bc_sink_create(#name, plist, buffer, role)
```

In this macro expansion, name is first used to represent a sink β-channel pointer, and secondly, used as the string #name, which can be converted to a unique integer tag by using an appropriate hash function [67, Section 2.9]. The sink-to-source link can then be established using these tags.

Implementation of algorithmic skeletons. We have shown in Section 4.4 an example implementation of the pipeline skeleton. By following similar approaches, we can implement several other skeletons, such as *farm*, *scan*, *map* etc., as a set of programming interfaces implemented on top of the β-channel interfaces. The advantage of such skeleton implementations, as discussed in Section 4.4.2, is that the

skeleton abstraction layer could be bypassed at runtime. This means that programmers can take advantage of skeletal programming during the application development phase, while avoiding possible performance degradation due to the skeleton abstraction layer overhead, since activation of the β-channels directly interacts with the low-level communication layer of the runtime system.

Integration with MPI. The β-channel runtime system is developed as an application programming library, which is independent of any MPI implementation. It is therefore possible for an application program to use both MPI and β-channel interfaces. Since the functional units of the runtime system (see Figure 5.1) exist independent of an MPI implementation, interfaces from either approach do not interfere with each other's functions.

APPENDIX A

Auxilliary functions

This appendix provides the source listing for the auxilliary functions utilised by the example implementations.

```
/* Bit complement. */
#define bit_complement ( num, bit ) ( ( 1 ≪ ( bit ) ) ⊗ ( num ) )
/* Complex data type */
typedef struct complex_s { double real; double img; } complex_t;
/* Complex addition. */
void complex_addition (complex_t *a, complex_t *b) {
   complex_t r;
   r.real := a→real + b→real; r.img := a→img + b→img;
   *a := r; /* Store the result. */
}
/* Complex addition. */
void complex_subtraction (complex_t *a, complex_t *b) {
   complex_t r;
   r.real := a→real − b→real; r.img := a→img − b→img;
   *a := r; /* Store the result. */
}
/* Complex multiplication. */
void complex_multiply (complex_t *r, complex_t *a, complex_t *b) {
   r→real := ( a→real * b→real ) − ( a→img * b→img );
   r→img := ( a→real * b→img ) + ( a→img * b→real );
}
/* Complex power. */
void complex_power (complex_t *r, complex_t *a, int p) {
   double c;
   int i;
   if (p = 0) { r→real := 1.0; r→img := 0.0; return; }
   r→real := a→real; r→img := a→img;
   for ( i := 1; i < p; i++ ) {
      c := ( r→real * a→real ) − ( r→img * a→img );
      r→img := ( r→real * a→img ) + ( r→img * a→real );
      r→real := c;
   }
}
/* Multiply with power of ith primitive root. */
void multiply_omega(complex_t *coeff, int i, int p) {
   complex_t a, b;
   double theta := 6.283185307 / i;
   a.real := cos ( theta );
   a.img := sin ( theta );
   complex_power ( &b, &a, p );
```

```
    complex_multiply (&a, coeff, &b );
    *coeff := a; /* Store the result. */
}
/* Compare and exchange. */
void compare_exchange (int nlocal, int *elem, int *workspace, short small) {
    int i, j, k;
    memcpy ( workspace, elem, bytes );
    if ( small ) {
        i := 0; j := nlocal;
        for ( k := 0; k < nlocal; k++ )
            if ( workspace[i] < workspace[j] ) elem[k] := workspace[i++];
            else elem[k] := workspace[j++];
    } else {
        i := local_upper; j := recv_upper;
        for ( k := i; k >= 0; k-- )
            if ( workspace[i] >= workspace[j] ) elem[k] := workspace[i--];
            else elem[k] := workspace[j--];
    }
}
/* Quick sort comparison. */
int compare (const void *x, const void *y) {
    return ( * ( int * ) x - * ( int * ) y );
}
/* Transpose a matrix. */
void transpose (int nrow, int ncol, int *matrix) {
    int *temp;
    int i, j, k := nrow * ncol;
    temp := (int *) calloc ( k, sizeof ( int ) );
    for ( i := 0; i < nrow; i++ )
        for ( j := 0; j < ncol; j++ ) {
            * ( temp + j * nrow + i ) := * ( matrix + i * ncol + j );
        }
    memcpy ( matrix, temp, k * sizeof ( int ) );
    free ( temp );
}
/* Matrix multiplication of blocks. */
void multiply_blocks (int *result, int nrblk, int ca, int *rows, int ncblk, int cb, int *cols, int n) {
    int i, j, *c, *temp;
    result += ( ( n + bc_rank ) % bc_size ) * ncblk;
    temp := result;
    for ( i := 0; i < nrblk; i++ ) {
        c := cols;
        for ( j := 0; j < ncblk; j++ ) {
            * ( temp + j ) := multiply_row_column ( rows, c, ca );
            c += ca;
        }
        rows += ca; temp += cb;
    }
}
/* Vector multiplication. */
```

```
int multiply_row_column (int *row, int *col, int c) {
   int result := 0, i;
   for ( i := 0; i < c; i++ ) result += * ( row + i ) * * ( col + i );
   return result;
}
/* Generate the XPM image file */
void generate_image (int *result, int rows, int cols) {
   FILE *out;
   int i, j, code;
   char colour[] := " .XoO+@#$%&*=-;:";
   out := fopen ( "mandelbrot.xpm", "w" );
   fprintf( out, "%s", "/* XPM file: Mandelbrot Set */\n"
         "static char *mandelbrot[] := {\n" );
   fprintf ( out, "\"%d %d 16 1\",\n", rows, cols );
   fprintf ( out, "%s", "\"  c #000000\",\n"
         "\". c #220000\",\n\"X c #440000\",\n\"o c #660000\",\n\"O c #880000\",\n"
         "\"+ c #aa0000\",\n\"@ c #cc0000\",\n\"# c #ee0000\",\n\"$ c #ee2200\",\n"
         "\"% c #ee4400\",\n\"& c #ee6600\",\n\"* c #ee8800\",\n\":= c #eeaa00\",\n"
         "\"- c #eebb00\",\n\"; c #eedd00\",\n\": c #eeff00\"");
#define SPREAD_COLLECT
   for (i := 0; i < rows; i++) {
      fprintf(out, "%s", ",\n\"");
      for ( j := 0; j < cols; j++ ) {
         code := * ( result + i * cols + j );
         fprintf ( out, "%c", colour[code%16] );
      }
      fprintf ( out, "%s", "\"" );
   }
   fprintf ( out, "%s", "};\n" );
#elif FARM_HARVEST
   for (i := 0; i < PIX_ROWS; i++) {
      fprintf(out, "%s", ",\n\"");
      for (k := 0; k < PIX_ROWS; k++)
         if (result[k].row = i) break;
      for (j := 0; j < PIX_COLS; j++) {
         code := result[k].color[j];
         fprintf(out, "%c", colour[code%16]);
      }
      fprintf(out, "%s", "\"");
   }
   fprintf(out, "%s", "};\n");
#endif
   fclose ( out );
}
/* Calculate set inclusion for a block of complex points. */
void calc_mandel_block ( int count, complex_t *in, int *out ) {
   int i;
   for ( i := 0; i < count; i++ ) out[i] := calc_mandel_pixel ( in[i] );
}
/* Calculate Set inclusion (returns color code). */
```

```
int calc_mandel_pixel ( complex_t c ) {
   int count := 0, max_iter := 255;
   complex_t z;
   double len_square, temp;
   z.real := z.img := 0.0;
   do {
      temp := z.real∗z.real − z.img∗z.img + c.real;
      z.img := 2.0∗z.real∗z.img + c.img; z.real := temp;
      len_square := z.real∗z.real + z.img∗z.img;
      if ( len_square > 4.0 ) break;
      count++;
   } while ( count < max_iter );
   return count;
}
```

Bibliography

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. [pp. 25 and 115.]

[2] Mohammad Mursalin Akon, Dhrubajyoti Goswami, and Hon Fung Li. A Model for Designing and Implementing Parallel Applications Using Extensible Architectural Skeletons. In *Proc. of PaCT*, volume 3606 of *LNCS*, pages 367–380. Springer-Verlag, 2005. [pp. 18.]

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976. [pp. 35.]

[4] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970. [pp. 35.]

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996. [pp. 14.]

[6] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Comput. Surv.*, 23(1):49–90, 1991. [pp. 2 and 13.]

[7] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Comput. Surv.*, 15(1):3–43, 1983. [pp. 12.]

[8] John Anvik, Jonathan Schaeffer, Duane Szafron, and Kai Tan. Why Not Use a Pattern-Based Parallel Programming System? In *Proc. of Euro-Par 2003*, volume 2790 of *LNCS*, pages 81–86, 2003. [pp. 18.]

[9] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995. [pp. 16.]

[10] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and Transformations in an Integrated Parallel Programming Environment. In V. Malyshkin, editor, *Proc. of PaCT*, volume 1662 of *LNCS*, pages 13–27. Springer-Verlag, 1999. [pp. 16.]

[11] Henri E. Bal. *Programming Distributed Systems*. Silicon Press, Summit, NJ, 1990. [pp. 14.]

[12] Henri E. Bal. A Comparative Study of Five Parallel Programming Languages. In *Proc. of EurOpen Spring Conf. on Open Distributed Systems*, pages 209–228, Tromso, 20–24 May 1991. [pp. 14.]

[13] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992. [pp. 14.]

[14] Vasanth Bala and Shlomo Kipnis. Process Groups: A Mechanism for the Coordination of and Communication Among Processes in the Venus Collective Communication Library. In *Proc. of IPPS*, pages 614–620, 1993. [pp. 2.]

[15] John Barnes. *Programming in Ada 95*. Addison Wesley, 1998. [pp. 123.]

[16] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *J. ACM*, 22(2):248–260, 1975. [pp. 132.]

[17] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible Skeletal Programming with eSkel. In *Proc. of Euro-Par*, volume 3648 of *LNCS*, pages 761–770. Springer-Verlag, 2005. [pp. 18.]

[18] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, second edition, 1998. [pp. 17.]

[19] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984. [pp. 13.]

[20] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distributed and abstract types in Emerald. *IEEE Trans. Softw. Eng.*, 13(1):65–76, 1987. [pp. 14.]

[21] G.H. Botorog and H. Kuchen. Skil: an Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. of the* 5^{th} *Intl. Symposium on High Performance Distributed Computing, Syracuse, New York, USA*, pages 243–252. IEEE Computer Society Press, 1996. [pp. 17.]

[22] Alex Brodsky, Jan Bækgaard Pedersen, and Alan Wagner. On the Complexity of Buffer Allocation in Message Passing Systems. *Journal of Parallel and Distributed Computing*, 65:692–713, 2005. [pp. 111.]

[23] Raymond M. Bryant, Anthony E. Krzesinski, M. Seetha Lakshmi, and K. Mani Chandy. The MVA priority approximation. *ACM Trans. Comput. Syst.*, 2(4):335–359, 1984. [pp. 132.]

[24] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proc. of Supercomputing Symposium*, pages 379–386, 1994. [pp. 145.]

[25] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. [pp. 112 and 145.]

[26] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, first edition, 1999. [pp. 11.]

[27] Jeffrey P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Commun. ACM*, 16(9):527–531, 1973. [pp. 132.]

[28] Lennart Carleson and Theodore W. Gamelin. *Complex Dynamics*. Springer-Verlag, 1993. [pp. 89.]

[29] K.M. Chandy, R. Manohar, B.L. Massingill, and D.I. Meiron. Integrating Task and Data Parallelism with the Group Communication Archetype. In *Proc. of* 9^{th} *Intl. Parallel Processing Symposium*, pages 724–733, Santa Barbara, CA, 1995. [pp. 18.]

[30] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001. [pp. 19.]

[31] W. D. Clinger. Foundations of Actor Sematics. Technical Report AI-TR-633, MIT Articificial Intelligence Laboratory, May 1981. [pp. 25 and 115.]

[32] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, Massachusetts, 1989. [pp. 7, 15 and 17.]

[33] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004. [pp. 7, 18 and 107.]

[34] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, April 1965. [pp. 83.]

[35] Robert Cypher and Eric Leu. The Semantics of Blocking and Nonblocking Send and Receive Primitives. In *Proc.* 18^{th} *IEEE Intl. Parallel Processing Symposium*, pages 729–735, April 1994. [pp. 121 and 122.]

[36] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P^3L. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of Euro-Par*, volume 1300 of *LNCS*, pages 619–628, Passau, Germany, August 1997. Springer-Verlag. [pp. 16.]

[37] J. Darlington, Y. Guo, and H. To. Functional Skeletons for Parallel Coordination. In *Proc. of Euro-Par*, volume 966 of *LNCS*, pages 55–66, August 1995. [pp. 17.]

[38] E.W. Felten and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proc. of Scalable High Performance Computing Conf.*, pages 84–89, Williamsburg, VA, 1992. [pp. 111.]

[39] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(0108):70–82, 1996. [pp. 111.]

[40] Ian T. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994. [pp. 12 and 20.]

[41] Nissim Francez and Brent Hailpern. Script: A communication abstraction mechanism. In *Proc. of the* 2^{nd} *ACM symposium on Principles of distributed computing*, pages 213–227, New York, NY, USA, 1983. ACM Press. [pp. 25.]

[42] Keir Fraser. *Practical Lock-freedom*. PhD thesis, University of Cambridge, 2004. [pp. 9.]

[43] L.L. Garlick. Out-of-band control signals in a host-to-host protocol. Network Working Group RFC 721, September 1976. [pp. 125.]

[44] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1995. [pp. 14.]

[45] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, January 1985. [pp. 13, 51 and 125.]

[46] Claudio Gennaro and Peter J. B. King. Parallelising the Mean Value Analysis Algorithm. *Trans. of the Society for computer Simulation Intl.*, 16(1):16–22, March 1999. [pp. 132.]

[47] Sergei Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Trans. on Programming Languages and Systems*, 26(1):47–56, 2004. [pp. 19.]

[48] Dhrubajyoti Goswami. *Parallel Architectural Skeletons: Re-usable Building Blocks for Parallel Applications*. Ph.d. thesis, University of Waterloo, Waterloo, Ontario, Canada, 2001. [pp. 18.]

[49] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003. [pp. 12, 79, 83, 97 and 122.]

[50] Lucio Grandinetti, editor. *Grid computing: the new frontier of high performance computing*, volume 14 of *Advances in parallel computing*. Elsevier, 2005. [pp. 11.]

[51] T. R. G. Green. Cognitive dimensions of notations. In *Proc. of the* 5th *Conf. of the British Computer Society Human-Computer Interaction Specialist Group*, People and Computers V, pages 443–460, Nottingham, 5–8 September 1989. British Computer Society, Cambridge University Press. [pp. 27.]

[52] T. R. G. Green. *Psychology of Programming*, chapter Programming Languages as Information Structures. Computers and People Series. Academic Press, 1990. [pp. 27.]

[53] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. [pp. 121.]

[54] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, second edition, 1999. [pp. 14.]

[55] William D. Gropp. Learning from the Success of MPI. In B. Monien, V.K. Prasanna, and S. Vajapeyam, editors, *Proc. of HiPC*, volume 2228 of *LNCS*, pages 81–92. Springer-Verlag, 2001. [pp. 4.]

[56] R. Guerraoui. Distributed Programming Abstractions. *ACM Comput. Surv.*, 28(4es):153, 1996. [pp. 13.]

[57] A. Gursoy and L.V. Kalé. Dagger: Combining the Benefits of Synchronous and Asynchronous Communication Styles. In H. G. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 590–596, Cancun, Mexico, April 1994. [pp. 51.]

[58] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. [pp. 1.]

[59] C. A. Herrmann and C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000. [pp. 17.]

[60] C. E. Hewitt and H. Baker. Laws for Communicating Parallel Processes. In *Proc. of the IFIP Congress*, pages 987–992, August 1977. [pp. 115.]

[61] C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University, Stanford Artificial Intelligence Laboratory, October 1973. [pp. 28.]

[62] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978. [pp. 13 and 164.]

[63] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, 1981. [pp. 28.]

[64] IEEE/ANSI. *9945-1:1996 (ISO/IEC) Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language] (ANSI)*. IEEE Standards Press, IEEE/ANSI Std. 1003.1 1996 edition, 1996. [pp. 60.]

[65] Wesley M. Johnston, J.R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004. [pp. 51.]

[66] R.M. Karp and R.E. Miller. Properties of a model for parallel computation: determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966. [pp. 111.]

[67] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999. [pp. 164.]

[68] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language: ANSI C*. Prentice-Hall Software Series. Prentice-Hall Inc., Second edition, 1988. [pp. 9 and 17.]

[69] Herbert Kuchen. A Skeleton Library. Technical Report 6/02-I, University of Münster, 2002. [pp. 18.]

[70] Koen Langendoen, Raoul Bhoedjang, and Henri Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, April–June 1997. [pp. 111.]

[71] Claudia Leopold. *Parallel and Distributed Computing: A survey of Models, Paradigms and approaches*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, John Wiley & Sons, Inc., 2001. [pp. 12, 13, 17 and 123.]

[72] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28:1663–1683, 2002. [pp. 18.]

[73] Spiros Mancoridis and Richard C. Holt. Recovering the structure of software systems using tube graph interconnection clustering. In *Proc. Intl. Conf. on Software Maintenance (ICSM)*, pages 23–32. IEEE, November 1996. [pp. 28.]

[74] B.L. Massingill and K.M. Chandy. Parallel program Archetypes. In *Proc.* 13$^{\text{th}}$ *Intl. Parallel and Distributed Processing (IPPS)*, pages 290–296, San Juan, 1999. [pp. 18.]

[75] Kiminori Matsuzaki, Kazuhiko Kakehi, and Hideya Iwasaki. A Fusion-Embedded Skeleton Library. In *Proc. Euro-Par*, volume 3149 of *LNCS*, pages 644–653. Springer-Verlag, 2004. [pp. 18.]

[76] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005. [pp. 12 and 18.]

[77] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models and Methods*. Prentice Hall, 2002. [pp. 133.]

[78] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992. [pp. 28.]

[79] Bertrand Meyer. Principles of language design and evolution. In *Millenial Perspectives in Computer Science (Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare)*, pages 229–246, Palgrave, Basingstoke-New York, 2000. [pp. 28.]

[80] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proc. of the* 3rd *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 18–28, New York, NY, USA, 1995. ACM Press. [pp. 28.]

[81] Ronald A. Olsson, Gregory R. Andrews, Michael H. Coffin, and Gregg M. Townsend. SR: A language for parallel and distributed programming. Technical Report TR 92-09, The University of Arizona, Tucson, 1992. [pp. 14.]

[82] C. Pair. *Psychology of Programming*, chapter Programming, Programming Languages and Programming Methods. Computers and People Series. Academic Press, 1990. [pp. 27.]

[83] H.-O. Peitgen and P.H. Richter. *The Beauty of Fractals: Images of Complex Dynamical Systems*. Springer-Verlag, 1986. [pp. 90.]

[84] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987. [pp. 27.]

[85] M. Petre. *Psychology of Programming*, chapter Expert Programmers and Programming Languages. Computers and People Series. Academic Press, 1990. [pp. 28.]

[86] Michael Jay Quinn. *Parallel Computing: Theory and Practice*. Computer Science Series. McGraw Hill International Edition, 1994. [pp. 12, 76, 79, 83, 85 and 97.]

[87] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91–118, 1991. [pp. 14.]

[88] Michel Raynal. *Distributed Algorithms and Protocols*. John Wiley and Sons, 1988. [pp. 24.]

[89] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980. [pp. 132.]

[90] Paul Sack and Anne C. Elster. Fast MPI Broadcast through Reliable Multicasting. In J. Fagerholm et al., editor, *Proc. PARA*, volume 2367 of *LNCS*, pages 445–453. Springer-Verlag, 2002. [pp. 32.]

[91] P. Schweitzer. Approximate analysis of Multiclass Closed Networks of Queues. In *Proc. Int. Conf. Stochastic Cont. Optimization*, Amsterdam, 1979. [pp. 132 and 133.]

[92] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998. [pp. 12.]

[93] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, 1996. [pp. 2, 12, 19, 21, 51, 126 and 152.]

[94] Richard W. Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming: The Sockets Networking API*, volume 1. Prentice Hall, third edition, 2003. [pp. 125 and 145.]

[95] Jaspal Subhlok, Peter Lieu, and Bruce Lowekamp. Automatic node selection for high performance applications on networks. In *Proc. 7th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 163–172, New York, NY, USA, 1999. ACM Press. [pp. 164.]

[96] Top500 supercomputer sites. http://www.top500.org/. [pp. 1 and 11.]

[97] Andrew S. Tanenbaum, M. Frank Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, 1992. [pp. 32.]

[98] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *Proc. 9th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 216–229, New York, NY, USA, 2003. ACM Press. [pp. 164.]

[99] Gregor von Bochmann. *Concepts for Distributed Systems Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983. [pp. 24.]

[100] Philip Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, 1998. [pp. 15.]

[101] Gregory V. Wilson. A Glossary of Parallel Computing Terminology. *IEEE Parallel & Distributed Technology*, 1(1):52–67, 1993. [pp. 32.]

[102] Nicklaus Wirth. *Algorithm + Data Structures = Programs*. Prentice-Hall series in Automatic Computation. Prentice-Hall, 1976. [pp. 59.]